ОГЛАВЛЕНИЕ

ББК 32.973
С 46
УДК 681.322-181.4

Редакция переводной литературы

Скэнлон Л.

С 46 Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера: Пер. с англ. – М. : Радио и связь. 1989 г. – 336 с. : ил.

ISBN 5-256-00300-3.

В книге автора из США рассмотрен широкий круг вопросов, связанных с программированием на языке ассемблера для персональных ЭВМ IBM PC и XT. Приведены полные системы команд микропроцессоров Intel 8088 и 8087. Рассмотрены вопросы использования ресурсов операционной системы DOS и управления внешними устройствами, работа с клавиатурой и звуковым генератором, программы обработки прерываний системы ввода-вывода. Приведено большое число примеров и задач с решениями.

Для программистов.

С $\frac{2404000000-155}{046(01)-89}$ 138-89 ББК 32.973

Производственное издание

СКЭНЛОН ЛЕО

ПЕРСОНАЛЬНЫЕ ЭВМ IBM PC и XT. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА

Заведующий редакцией Ю. Г. Ивашов
Редактор М. Г. Коробочкина
Художественный редактор А. С. Широков
Переплет художника Б. И. Никопашина
Технический редактор Л. А. Горшкова
Корректор Н. Л. Жукова

ИБ № 1638

Подписано в печать с оригинал-макета 31.08.89. Формат 70 × 100/16. Бумага офс. № 2. Гарнитура пресс-роман. Печать офсетная. Усл. печ. л. 27,3. Усл. кр.-отт. 27,3. Уч.-изд. л. 25,60. Тираж 50 000 экз. Изд. № 22 384. Зак. №2434 Цена 2 р.
Издательство "Радио и связь". 101000, Москва, Почтамт, а/я 693

Московская типография № 4 Союзполиграфпрома при Государственном комитете СССР по печати. 129041, Москва, Б. Переяславская ул., д. 46

ISBN 5-256-00300-3 (рус.)
ISBN 0-89303-575-0 (англ.)

ПРЕДИСЛОВИЕ К РУССКОМУ ИЗДАНИЮ

Принято считать, что знание языка ассемблера необходимо лишь профессиональным программистам. Это подкрепляется тем, что большинство научно-инженерных и экономических задач гораздо легче программировать на языках высокого уровня, таких как Бейсик, Фортран, Паскаль, а наличие огромного числа готовых прикладных программ для персональных ЭВМ IBM PC и XT позволяет во многих случаях вообще обойтись без программирования в обычном смысле этого слова.

Тем не менее интерес к языку ассемблера для персональных ЭВМ IBM PC и XT постоянно растет, и не только среди профессиональных программистов. Объясняется это тем, что для самостоятельного выполнения простых доработок готовых программ и дополнения возможностей, предоставляемых трансляторами языков программирования высокого уровня, существует целый арсенал популярных и недорогих средств, таких как программы анализа и редактирования двоичных файлов, отладчики и ловушки прерываний и т. д. Грамотное применение этих средств сокращает затраты времени на доработку и отладку программ, а также повышает эффективность использования персональных ЭВМ. Однако для этого пользователю необходимо иметь определенный минимум знаний языка ассемблера, а также структуры и функции операционной системы PC/DOS.

Предлагаемая читателю книга Л. Скэнлона рассчитана на предоставление подобного минимума знаний, позволяющего уверенно выполнять небольшие ассемблерные вставки в программах на языках Бейсик, Фортран и Паскаль, а также пользоваться пакетами программ типа Norton Utilities и PC Tools для несложного редактирования исполняемых программных файлов (например, для замены или перевода на русский язык выдаваемых ими сообщений). Эту книгу можно рекомендовать широкому кругу пользователей персональных ЭВМ, совместимых с IBM PC и XT. Кроме того, ее можно рассматривать как элементарное введение в операционную систему PC DOS и язык ассемблера для персональных ЭВМ IBM PC и XT, вполне доступное старшеклассникам и студентам вузов.

И. В. Емелин

ПРЕДИСЛОВИЕ

ЗАЧЕМ НУЖЕН ЯЗЫК АССЕМБЛЕРА?

Многие люди пишут все свои программы на одном из так называемых языков программирования высокого уровня, чаще всего на языке Бейсик. Его легко выучить, он прост в применении и обеспечивает достаточное быстродействие при решении многих вычислительных задач. Зачем же тогда нужны другие языки?

Одна из причин их создания состоит в том, что Бейсик, как и обычные языки, не одинаково пригоден для всех задач. Например, представьте себе попытку описать кулинарное искусство без использования слов французского происхождения или симфоническую музыку без терминов на итальянском.

Аналогично некоторые специальные приложения ЭВМ, например компьютерная графика или музыка, обработка текстов, гораздо легче реализуются с помощью специальных языков.

Далее, Бейсик довольно медлителен. Это может удивить новичка, поскольку короткие программы исполняются почти мгновенно. Однако затруднения возникают в следующих ситуациях:

1. Обработка большого числа данных. Обратите внимание, насколько медлителен Бейсик, если, например, программа должна отсортировать длинный перечень фамилий и адресов или счетов. Бейсик также довольно медлителен, если программа должна найти нужное место в 50-страничном отчете или обновить инвентаризационную ведомость с тысячами наименований;

2. Построение графиков. Если программа изображает на экране рисунок, то либо она должна работать быстро, либо задержка окажется невыносимой. Если элементы рисунка должны двигаться, то программа должна выполняться достаточно быстро, чтобы движение выглядело естественным. Это особенно трудно, если рисунок состоит из многих элементов (например, космических кораблей, космических станций и инопланетян-захватчиков), которые должны двигаться в различных направлениях;

3. Большое число решений или "размышлений". Они часто требуются в сложных играх, например при игре в шашки или шахматы. Программа должна просматривать много вариантов, и чем сложнее анализ, тем больше времени требуется ЭВМ, чтобы сделать ход.

Почему Бейсик медлителен? Прежде всего из-за того, что в действительности ЭВМ транслирует каждый оператор Бейсика в простые внутренние команды (на так называемом машинном языке или языке ассемблера). Она делает это при каждом исполнении программы на Бейсике. Таким образом, большую часть времени ЭВМ тратит не на исполнение программы, а на ее трансляцию.

Есть версии Бейсика (называемые *компиляторами*), которые выполняют трансляцию один раз, а затем сохраняют оттранслированную версию. Однако и в этом случае Бейсик оказывается медлительным из-за своей механистичности. Действительно, он похож на автомобиль с автоматической коробкой передач. Мыслящее существо действует более гибко, более искусно и более хитро, чем автоматическая коробка передач либо компилятор или интерпретатор языка Бейсик.

Язык ассемблера представляет собой для ЭВМ эквивалент ручной коробки пе-

редач. Он обеспечивает программисту больший контроль над ЭВМ за счет большего труда, большей детализации и меньшего удобства. Подобно автоматической коробке передач Бейсик достаточно хорош в большинстве случаев для большинства программистов. Но тем, кому необходимо добиться максимальной производительности от своих ЭВМ, необходим язык ассемблера. Вы обнаружите, что большинство сложных игровых программ, программ для изображения рисунков и больших программ для экономических расчетов, по крайней мере частично, написаны на языке ассемблера.

Даже если язык ассемблера и покажется Вам подходящим, могут появиться сомнения, достаточно ли у Вас навыков для освоения программирования на языке ассемблера. Если Вам уже приходилось иметь дело с какого-либо рода программированием, то этого достаточно. Если Вы знаете Бейсик или какой-либо иной язык программирования высокого уровня, то это хорошо. Если Вам приходилось разрабатывать программы на языке ассемблера, то это еще лучше. Для читателей, у которых есть опыт программирования на языке высокого уровня, эта книга имеет две отправные точки.

Если Вы никогда не программировали на языке ассемблера, начните с введения, которое даст Вам "интенсивный курс" *двоичной* и *шестнадцатеричной* систем счисления. Но если Вы уже знаете, что означают эти термины, и понимаете, как эти системы использовать, то начните сразу с гл. 1.

СОДЕРЖАНИЕ ЭТОЙ КНИГИ

В гл. 1 мы дадим общее представление о микропроцессоре 8088 – "мозге" персональной ЭВМ IBM PC – и обсудим его роль в системе.

В начале гл. 2 обсуждаются общие свойства Ассемблеров, а затем дается описание *Макроассемблера* фирмы IBM. Все Ассемблеры PC-совместимых ЭВМ выполняют одну и ту же работу: они транслируют понятные Вам команды в числовые коды, которые понятны микропроцессору ЭВМ. Следовательно, большинство из описываемых нами принципов применимы к любому другому Ассемблеру, который может оказаться в Вашем распоряжении. В гл. 2 представлена также простая программа и показано, как ее ввести в ЭВМ, оттранслировать и исполнить (вызвать).

В гл. 3 описана система команд микропроцессора 8088, т. е. команды языка ассемблера, с помощью которых обеспечивается взаимодействие с Вашей ЭВМ IBM PC. Команды в этой книге описаны не в алфавитном порядке, а в зависимости от их принадлежности к функциональным группам. Другими словами, мы группируем сложение с вычитанием, умножение с делением и т. д. При таком подходе Вы не только *понимаете*, что делают команды, но и получаете представление об их взаимосвязях.

В гл. 4 показано, какие сочетания команд надо использовать для выполнения довольно сложных математических операций, которые в системе команд микропроцессора непосредственно не предусмотрены. В гл. 5 обсуждаются операции над списками и таблицами.

Персональные ЭВМ IBM PC и XT имеют встроенную управляющую программу, называемую BIOS (Basic I / O System – основная система ввода-вывода), которая обеспечивает взаимодействие с оборудованием системы. Другими словами, она обеспечивает все необходимое, что требуется для взаимодействия микропроцессора с клавиатурой, экраном, дисководом, принтером и другими периферийными устройствами. Следовательно, система BIOS выполняет функции "главного администратора" ЭВМ и обладает многими полезными возможностями, знание которых позволит Вам сберечь часы, затрачиваемые на программирование. В гл. 6 показано, как реализовать эти возможности.

В гл. 7, 8 обсуждаются программы, которые изображают на экране простые графические образы и генерируют звуки (даже музыку !) с помощью динамика, встроенного в ЭВМ IBM PC.

В гл. 9 рассмотрены *макроопределения*. Макроопределение представляет собой *мини*-программу, которая помещается в текст основной программы просто упоминанием имени. Применение макроопределений может упростить разработку программ на языке ассемблера до уровня программ на языке Бейсик. В этой главе приводится также "библиотека" из более чем 30 полезных макроопределений.

В гл. 10 описано использование *объектных библиотек*, т. е. дисковых файлов, содержащих уже оттранслированные программы. Фактически объектная библиотека представляет собой набор готовых к употреблению инструментов, которые Ваша программа может выбрать по мере необходимости. Возможность создания объектных библиотек введена фирмой IBM в Макроассемблер версии 2.

В гл. 11 описано еще одно новое свойство Макроассемблера версии 2:он позволяет Вам разрабатывать *структурированные* программы на языке ассемблера. Под "структурированной" мы понимаем программу, содержащую операторы высокого уровня, подобные операторам IF-THEN и FOR-NEXT в Бейсике. Структурирование ускорит Вашу работу над программой и сделает ее более легкой для понимания.

Наконец в гл. 12 мы обсудим математический сопроцессор 8087,представляющий собой дополнительную микросхему, которая может выполнять сложные арифметические операции.

Для удобства читателя в книгу включены пять приложений. В приложении А даны таблицы, которые помогут преобразовывать шестнадцатеричные числа в десятичные и наоборот. В приложении Б показаны символы и образы, которые можно изобразить на экране. В приложениях В и Г в алфавитном порядке перечислен набор команд микропроцессора 8088 и показано, сколько времени занимает выполнение каждой из команд, сколько байтов памяти занимает каждая команда и на какие флаги состояния она воздействует. Приложение Д представляет собой "руководство по пользованию диском", в котором описано, как использовать программы, находящиеся на дополнительно поставляемом диске.

Большинство глав завершается упражнениями. Некоторые из них помогут проверить, насколько Вы поняли материал главы, другие дополнят Ваше знание материала.

ЧТО ТРЕБУЕТСЯ ПРИ ЧТЕНИИ ЭТОЙ КНИГИ

Для работы над книгой Вам нужна персональная ЭВМ IBM PC, имеющая по крайней мере один дисковод, или персональная ЭВМ IBM XT. Вам потребуются также два пакета программ: *Ассемблер и дисковая операционная система DOS* (Disk Operating System) фирмы IBM.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Эта книга построена как дополнение к руководству по Макроассемблеру и руководствам, поставляемым вместе с ЭВМ, поэтому, скорее всего, другая литература Вам не понадобится. Однако всякий серьезный пользователь ЭВМ IBM PC должен иметь копию технического руководства (Technical Reference) по персональной ЭВМ IBM PC (или XT),представляющего собой бесценный источник технических деталей. Наряду с прочим материалом это руководство содержит полный и хорошо документированный листинг встроенной в ЭВМ управляющей программы BIOS.

Если Ваша ЭВМ снабжена операционной системой DOS версии 2.1 (или более

поздней), то полезно приобрести копию технического руководства по операционной системе DOS фирмы IBM (DOS Technical Reference), содержащего детальные сведения об организации хранения данных на диске, о процедурах DOS, к которым можно обращаться из программ на языке ассемблера, и о других возможностях системы.

Наконец, для получения детальных сведений об интегральных микросхемах системы Вам могут понадобиться следующие справочные документы: The iAPX 86, 88 User's Manual; The 8086 Family User's Manual; iAPX 88 Book; iAPX 88/10 Data Sheet. Их можно заказать в литературном отделе фирмы Intel Corporation по адресу 3065 Bowers Ave. ; Santa Clara, CA 95051.

ИЗМЕНЕНИЯ В ЭТОМ ИЗДАНИИ

Обладатели предыдущих изданий этой книги, наверное, обратят внимание на некоторые изменения, сделанные в данном издании. Некоторые из них отражают новые функции, предусмотренные в Макроассемблере версии 2.00; к ним относятся новые главы, посвященные объектным библиотекам и структурному программированию (гл. 10,11). Кроме того, автором добавлена глава, описывающая макроопределения (гл. 9), а также расширены и дополнены многие разделы с учетом личного опыта и предложений читателей. Короче говоря, автор попытался сделать эту книгу более полезной как для чтения, так и для справок. Язык ассемблера представляет собой очаровательное и эффективное средство программирования, и автор надеется, что читатель получит столько же удовлетворения от его использования, сколько получил автор от написания этой книги.

Лео Д. Скэнлон,
июнь 1985 г.

Эта книга посвящается моим сыновьям,
Роджеру и Райану,
чье поколение совершит "чудеса"
в 21-м веке.

ВВЕДЕНИЕ. ИНТЕНСИВНЫЙ КУРС СЧИСЛЕНИЯ ДЛЯ ЭВМ.

Если только Вы не прибыли с другой планеты, то всю жизнь при подсчете предметов Вы пользуетесь десятичными числами. Десятичная система является системой счисления по *основанию 10*, т. е.в ней есть 10 цифр от 0 до 9.

Для людей очень удобна десятичная система (вероятно, потому, что у нас по 10 пальцев на руках и ногах), но для ЭВМ она неудобна. Вместо этого ЭВМ использует систему счисления по основанию 2 (или *двоичную* систему), в которой есть только две цифры, 0 и 1. Следовательно, чтобы общаться с ЭВМ на ее языке (что приходится делать при программировании на языке ассемблера), Вы должны быть знакомы с двоичной системой. При программировании на языке ассемблера программисты применяют еще одну систему счисления – по основанию 16 (или *шестнадцатеричную*), поэтому Вы должны быть знакомы и с ней.

Эта глава представлят собой интенсивный курс систем счисления для ЭВМ и адресована тем читателям, которые ранее с ними не сталкивались. Если Вы уже разбираетесь в двоичной и шестнадцатеричной системах счисления, то можете пропустить эту главу и начать с гл. 1.

В. 1. ДВОИЧНАЯ СИСТЕМА СЧИСЛЕНИЯ

ЭВМ берет все команды программы и все данные из *памяти*. Память образована интегральными микросхемами (или "чипами"), которые содержат тысячи электронных компонентов. Подобно выключателям освещения эти компоненты имеют только два возможных состояния: "включен" или "выключен", с помощью которых комбинации компонентов памяти могут представлять числа любой длины. Каким образом?

Состояния компонентов памяти "включен" и "выключен" соответствуют двум цифрам *двоичной системы счисления*, фундаментальной системы счисления для ЭВМ, где 1 означает "включен", а 0 – "выключен". Конечно, эта система отличается от традиционной десятичной системы счисления.

Эти похожие на переключатели компоненты памяти называются *битами*, от сокращения binary digit (bit). По принятому соглашению "включенный" бит имеет значение 1, а "выключенный" – 0. Может показатся, что это сильно ограничивает возможности ЭВМ, но вспомните, что десятичные цифры заполняют всего лишь диапазон от 0 до 9. В точности так же, как Вы комбинируете десятичные цифры для получения чисел, превышающих 9, можно образовать из двоичных цифр числа, превышающие 1.

Как Вам известно, для представления десятичных чисел, превышающих 9, требуется дополнительная цифра в позиции "десятков". Аналогично для представления десятичных чисел, превышающих 99, требуется дополнительная цифра в позиции "сотен" и т. д. Каждая добавляемая Вами десятичная цифра имеет *вес*, в десять раз больший, чем соседняя справа цифра.

Например, Вы можете представить десятичное число 324 как

$(3 \cdot 100) + (2 \cdot 10) + (4 \cdot 1)$ или $(3 \cdot 10^2) + (2 \cdot 10^1) + (4 \cdot 10^0)$.

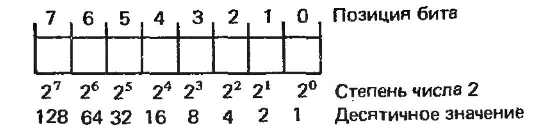


Рис. В.1. Веса восьми двоичных цифр

Или более математически: степень десяти при каждой десятичной цифре на единицу больше, чем у предшествующей цифры.

Аналогичные правила применимы к двоичной системе счисления. В этом случае *степень двойки при каждой двоичной цифре на единицу больше, чем у предшествующей цифры.* Крайний правый бит (двоичный разряд) имеет вес 2^0 (десятичное значение 1), следующий бит – вес 2^1 (десятичное значение 2) и т. д. Например, двоичное число 101 имеет десятичное значение 5, поскольку

$$101_2 = (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) = (1 \cdot 4) + (0 \cdot 2) + (1 \cdot 1) = 5_{10}\ .$$

Теперь Вы понимаете, как конструируются двоичные числа? Чтобы определить значение любой заданной позиции бита, надо удвоить вес предшествующей позиции. Так, десятичные веса первых восьми битов равны 1, 2, 4, 8, 16, 32, 64 и 128. Они показаны на рис. В.1.

Для преобразования десятичных чисел в двоичные надо сделать ряд вычитаний. Каждое вычитание даст значение отдельной двоичной цифры (бита).

Прежде всего вычтите из десятичного числа наибольший возможный двоичный вес и запишите 1 в *эту* позицию бита. Затем вычтите из результата новый наибольший возможный двоичный вес и запишите 1 в эту новую позицию бита. Продолжайте до получения нулевого результата. Запишите 0 во все те позиции битов, чьи веса не вычитались из текущего десятичного значения. Например, для преобразования десятичного числа 50 в двоичное надо выполнить следующие действия:

```
 50
-32  (позиция бита 5=1)
 18
-16  (позиция бита 4=1)
  2
 -2  (позиция бита 1=1).
  0
```

Записывая 0 в остальные позиции битов (биты 3, 2 и 0), получаем окончательный результат 110010.

Чтобы убедиться в том, что двоичный эквивалент десятичного числа 50 действительно равен 110010, сложим десятичные веса тех позиций, в которых стоит 1:

```
 32  (бит 5)
 16  (бит 4)
+ 2  (бит 1)
 50
```

ВОСЕМЬ БИТОВ ОБРАЗУЮТ БАЙТ

Персональная ЭВМ Apple II, Commodore 64, TRS-80 фирмы Radio Shack и многие другие сконструированы на базе 8-битовых *микропроцессоров,* названных так потому, что они могут обрабатывать по восемь битов информации за один прием.

Для обработки более восьми битов они должны выполнить дополнительные операции.

В принятой для ЭВМ терминологии 8-битовая единица информации называется *байтом*. Имея восемь битов, байт может представлять десятичные значения от 0 (двоичное 00000000) до 255 (двоичное 11111111).

Поскольку байт является основной единицей обработки, то микропроцессоры описываются в терминах числа байтов (а не битов), которые может иметь их память. Производители микроЭВМ конструируют память из блоков по 1024 байта. Это конкретное значение отражает двоичную ориентацию ЭВМ, поскольку представляет собой 2^{10} байтов.

Значение 1024 имеет стандартное сокращение: букву К. Следовательно, ЭВМ, имеющая "48К памяти", содержит 48·1024 (или 49 152) байтов.

СЛОЖЕНИЕ ДВОИЧНЫХ ЧИСЕЛ

Вы можете складывать двоичные числа тем же способом, что и десятичные: путем переноса любого избытка из одного столбца в следующий. Например, при сложении десятичных значений 7 и 9 надо перенести 1 в позицию десятков, что обеспечит получение правильного результата – 16. Аналогично при сложении двоичных значений 1 и 1 Вы должны перенести 1 в позицию "двоек", что обеспечит правильный результат – 10.

Сложение чисел, записанных многими битами, несколько сложнее, поскольку при этом приходится учитывать перенос из предыдущего столбца. Например, при выполнении следующей операции происходит два переноса:

```
 1011
+  11
-----
 1110.
```

Сложение в крайнем правом столбце (1+1) дает результат 0 и перенос 1 во второй столбец. С учетом этого переноса сложение во втором столбце (1+1+1) дает результат 1 и перенос 1 в третий столбец.

Общие правила двоичного сложения показаны в следующей таблице:

Исходные данные			Результат	
Операнд 1	Операнд 2	Перенос	Сумма	Перенос
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

ЧИСЛА СО ЗНАКОМ

До сих пор мы обсуждали двоичное представление *чисел без знака.* Как уже упоминалось, у числа без знака каждый бит имеет вес, зависящий от его позиции. Крайний правый (или младший) бит имеет вес 1, а каждый следующий бит имеет вес, вдвое больший, чем предшествующий ему бит. Следовательно, если все восемь битов байта равны 0, то байт имеет значение 0; если все они равны 1, то байт имеет значение 255.

Однако вычисления могут содержать как положительные, так и отрицательные значения, другими словами, *числа со знаком.* Если байт содержит число со знаком, то его значение представляется только младшими семью битами (0 – 6); старший бит (бит 7) указывает знак числа. *Знаковый бит равен 0, если число положительно или равно 0, и 1, если оно отрицательно.* На рис. В.2 показано размещение битов в байтах со знаком и без знака.

Если байт содержит числа со знаком, то он может представлять положительные значения от 0 (двоичное значение 00000000) до +127 (двоичное значение 01111111) и отрицательные значения от –1 (двоичное значение 11111111) до –128 (двоичное значение 10000000).

ПРЕДСТАВЛЕНИЕ ЗНАЧЕНИЙ В ДОПОЛНИТЕЛЬНОМ КОДЕ

Почему –1 представляется в двоичном виде как 11111111, а не 10000001? Дело в том, что отрицательные числа со знаком представляются в *дополнительном коде* (в форме дополнения до двух). Ученые-программисты ввели дополнительный код, чтобы нуль не имел два различных представления: все нули ("положительный нуль") и все нули с 1 в позиции знака ("отрицательный нуль").

Чтобы найти двоичное представление отрицательного числа (т.е. его дополнительный код), надо просто взять его положительную форму, обратить каждый бит (т.е. заменить 1 на 0, а 0 на 1), а затем добавить к полученному результату 1.

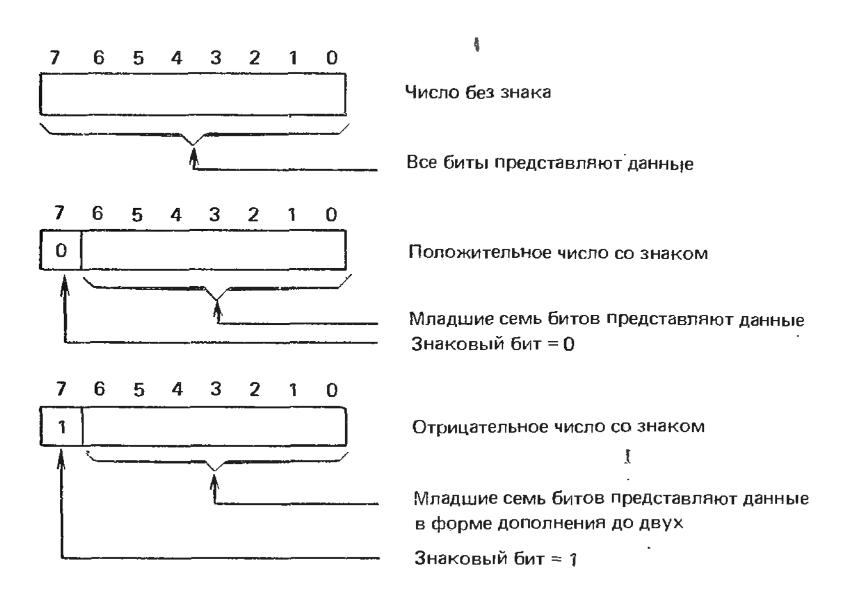


Рис. В.2. Представление чисел со знаком и без знака

Следующий пример показывает, как вычислить двоичное представление числа –32 в дополнительном коде:

```
  00100000   + 32

  11011111   (обратить каждый бит)
+        1   (добавить 1)
  11100000   (дополнительный код).
```

Конечно, использование дополнительного кода приводит к тому, что отрицательное число становится трудно расшифровать. К счастью, только что описанной процедурой можно воспользоваться для того, чтобы получить положительную форму отрицательного числа (записанного в дополнительном коде). Например, чтобы узнать, какое значение имеет число 11010001, сделайте следующее:

```
  00101111   (обратить каждый бит)
+        1   (добавить 1)
  00110000   ( = 16 + 32 = + 48).
```

Программы на языке ассемблера позволяют Вам вводить числа в десятичном виде (со знаком и без знака) и автоматически выполняют все преобразования. Однако иногда Вам может понадобиться интерпретировать отрицательное число, содержащееся в памяти или в регистре, поэтому не мешает знать, как самим выполнить эти преобразования.

В.2. ШЕСТНАДЦАТЕРИЧНАЯ СИСТЕМА СЧИСЛЕНИЯ

Хотя двоичная система и обеспечивает точное представление чисел в памяти, тем не менее с последовательностью из одних нулей и единиц трудно работать. Кроме того, возрастает вероятность совершить ошибку, поскольку при наборе числа вида 10110101 чрезвычайно легко сделать опечатку.

Много лет тому назад программисты убедились, что обычно им приходилось работать не с отдельными битами, а с *группами* битов. Первые микропроцессоры были 4-битовыми устройствами (они обрабатывали по четыре бита за один прием), поэтому логической альтернативой двоичной системе оказалась система, которая оперировала четверками битов.

Как Вам известно, четырьмя битами можно представить двоичные значения от 0000 до 1111 (что эквивалентно десятичным значениям от 0 до 15), т.е. всего 16 возможных комбинаций. Если в системе счисления должны быть обозначены все эти комбинации, то она должна иметь 16 цифр. Другими словами, это должна быть система счисления по *основанию 16.*

Если "двоичная" означает систему по основанию 2, а "десятичная" – систему по основанию 10, то каким термином назвать систему по основанию 16? Соединив греческое слово hex (шесть) и латинское слово decem (десять), получили слово hexadecimal (шестнадцатеричный). Следовательно, система счисления по основанию 16 называется *шестнадцатеричной системой.*

Из 16 цифр шестнадцатеричной системы счисления первые 10 получили обозначения от 0 до 9 (десятичные значения от 0 до 9), а остальные шесть – от A до F (десятичные значения от 10 до 15). В табл. В.1 перечислены двоичные и десятичные значения каждой шестнадцатеричной цифры.

Подобно двоичным и десятичным цифрам каждая шестнадцатеричная цифра имеет вес, кратный основанию счисления. Так как шестнадцатеричная система

Таблица В. 1. **Шестнадцатеричная система счисления**

Шестнадцатеричная цифра	Двоичное значение	Десятичное значение	Шестнадцатеричная цифра	Двоичное значение	Десятичное значение
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

счисления построена по основанию 16, то каждая цифра имеет вес, в 16 раз больший, чем соседняя справа цифра. Таким образом, крайняя правая цифра имеет вес 16^0, следующая – вес 16^1 и т.д. Например, шестнадцатеричное значение 3AF имеет десятичное значение 943, поскольку запись

$$(3 \cdot 16^2) + (A \cdot 16^1) + (F \cdot 16^0)$$

в десятичной форме приобретает вид

$$(3 \cdot 256) + (10 \cdot 16) + (15 \cdot 1) = 943.$$

ПРИМЕНЕНИЕ ШЕСТНАДЦАТЕРИЧНЫХ ЧИСЕЛ

Если в языке Бейсик и других языках программирования высокого уровня числа обычно изображаются в десятичном виде, то в языке ассемблера они обычно изображаются в шестнадцатеричном виде. Это относится к адресам, кодам команд и содержимому ячеек памяти и регистров. Следовательно, чтобы извлечь максимум пользы из своей работы над программами, старайтесь "думать по-шестнадцатеричному". Поначалу это трудно, но по мере появления опыта становится все легче и легче.

Чтобы помочь в этом, в приложении А приводятся таблицы преобразования десятичных чисел в шестнадцатеричные и обратно. Если Вам не хочется связываться с таблицами, то на поставляемом с книгой диске содержится программа HEX2DEC.EXE, которая преобразует шестнадцатеричное число (до четырех цифр) в десятичное – как со знаком, так и без знака. Для ее вызова вставьте диск стороной 2 (объектный код) вверх, затем наберите hex2dec и нажмите клавишу возврата каретки. Для завершения работы с программой нажмите клавишу возврата каретки в ответ на приглашение к вводу.

УПРАЖНЕНИЯ

1. Преобразуйте следующие десятичные значения в двоичные:
   а) 12; б) 17; в) 45; г) 72.
2. Преобразуйте следующие двоичные значения без знака в десятичные:
   а) 1000; б) 10101; в) 11111.
3. Как бы Вы записали три двоичных числа из упр. 2 в шестнадцатеричном виде?
4. Укажите десятичный эквивалент шестнадцатеричного числа D8, если:
   а) D8 представляет число без знака;
   б) D8 представляет число со знаком.

ГЛАВА 1. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА

1. 1. ЧТО ТАКОЕ ЯЗЫК АССЕМБЛЕРА?

Подобно Бейсику язык ассемблера представляет собой набор слов, задающих ЭВМ действия, которые она должна выполнить. Но в отличие от языка Бейсик слова из набора команд языка ассемблера имеют непосредственное отношение к компонентам ЭВМ. Описания действий ЭВМ, данные на языке Бейсик и на языке ассемблера, связаны между собой так же, как простое указание повернуть за угол и детальное описание процесса сокращений мышц и маневрирования при обходе препятствия. Обычному человеку в большинстве случаев достаточно простого указания; более детальные команды могут понадобиться только атлетам или альпинистам.

Программы, написанные на языке ассемблера, дают ЭВМ более детальные команды, например "загрузить в регистр AX значение 32", "поместить в регистр DL содержимое регистра CL", "запомнить содержимое регистра DL в ячейке памяти с адресом 3456". Как видите, язык Бейсик и язык ассемблера различаются способом задания действий ЭВМ: *на Бейсике Вы даете общие указания, на языке ассемблера – конкретные.*

Хотя программы на языке ассемблера пишутся дольше, чем аналогичные программы на Бейсике, они гораздо быстрее выполняются. Причиной этому служит уровень детализации команд. Здесь уместна аналогия с атлетом, который достигает лучших успехов в беге или прыжках за счет контроля каждого своего движения. Для достижения большей производительности существенна точность выполняемых действий.

Так как язык ассемблера требует от Вас задания действий на уровне внутренних компонентов ЭВМ, то Вам надо понимать свойства и возможности интегральной микросхемы, содержащей эти компоненты, а именно микропроцессора ЭВМ. В этой книге мы будем иметь дело с микропроцессором Intel 8088, работающим в персональных ЭВМ IBM PC/XT. Прежде чем описывать устройство этого микропроцессора, коснемся истории его появления на свет.

1. 2. ПРОИСХОЖДЕНИЕ МИКРОПРОЦЕССОРА 8088

Первые микропроцессоры были 4-битовыми устройствами. Это означает, что за один прием они могли обрабатывать только четыре бита информации. Для обработки более четырех битов они должны были выполнить несколько последовательных операций. Конечно, это замедляло работу.

Первым промышленным *8-битовым* микропроцессором (обрабатывавшим одновременно 8 битов информации) стал микропроцессор 8008, выпущенный фирмой Intel в 1972 году. Он считается лучшим 8-битовым микропроцессором первого поколения. По своей архитектуре этот микропроцессор похож на калькулятор; он имеет аккумулятор, шесть регистров общего назначения, указатель стека (специальный регистр адреса рабочих ячеек), восемь регистров адреса и специальные команды для ввода и вывода данных. В 1973 году фирма Intel выпустила версию второго поколения микропроцессора 8008, получившую название 8080.

По сравнению с микропроцессором 8008 микропроцессор 8080 мог адресоваться к большей памяти, имел расширенные возможности ввода-вывода, обладал дополнительными командами и работал быстрее. Хотя идеи архитектуры микропроцессора 8008 были в основном перенесены фирмой Intel и на микропроцессор 8080, его внутренняя организация была улучшена настолько, что микропроцессор 8080 стал стандартом de facto для микропроцессоров второго поколения; и когда речь заходит о микропроцессорах, то многим людям первым делом приходит на ум именно микропроцессор 8080.

Достижения технологии позволили фирме Intel в 1976 году выпустить усовершенствованную версию микропроцессора 8080, названную 8085. Он отличался от микропроцессора 8080 конструкцией корпуса, имел сброс в начальное состояние (для инициализации микропроцессора), прерывания по вектору (для обслуживания периферийных устройств), последовательный порт ввода-вывода (для подключения принтеров и других периферийных устройств). Кроме того, ему требовался только один источник питания +5 В (микропроцессору 8080 требовалось два источника питания).

Ко времени выпуска микропроцессора 8085 фирма Intel встретилась с серьезной конкуренцией на рынке 8-битовых микропроцессоров. Начали пользоваться успехом микропроцессор Z80 фирмы Zilog Corporation, представляющий собой усовершенствование микропроцессора 8080, а также микропроцессоры 6800 фирмы Motorola и 6502 фирмы MOS Technology (ныне фирма Commodore), существенно отличавшиеся от 8080 по своей архитектуре. И вместо того, чтобы продолжать борьбу на переполненном рынке 8-битовых микропроцессоров, фирма Intel сделала качественный шаг вперед и в 1978 году выпустила 16-битовый микропроцессор 8086, который мог обрабатывать данные в 10 раз быстрее, чем микропроцессор 8080.

Микропроцессор 8086 программно совместим с микропроцессором 8080 на уровне языка ассемблера. Это означает, что после некоторых минимальных преобразований программы, написанные для микропроцессора 8080, можно заново оттранслировать и выполнить на микропроцессоре 8086. Такая совместимость обеспечивается за счет того, что регистры микропроцессора 8080 и набор его команд являются как бы подмножествами регистров и набора команд микропроцессора 8086. Это позволило фирме Intel воспользоваться богатым опытом применения микропроцессора 8080 и получить преимущество в сфере более сложных приложений.

Так как многие проектировщики все еще предпочитали пользоваться в своих 16-битовых системах более дешевыми 8-битовыми вспомогательными и периферийными микросхемами, то фирма Intel выпустила версию микропроцессора 8086, имевшую то же устройство внутри, но 8-битовую шину данных снаружи. Эта версия (микропроцессор 8088) идентична 8086, но затрачивает больше времени на передачи 16-битовых данных, которые выполняются с помощью двух последовательных 8-битовых передач. Однако в приложениях, где обрабатываются в основном 8-битовые значения, микропроцессор 8088 отстает по производительности от микропроцессора 8086 не более чем на 10%.

Таким образом, Вы можете считать, что Ваша персональная ЭВМ фирмы IBM имеет 16-битовый микропроцессор. (И, следовательно, можете пользоваться многочисленной литературой по микропроцессору 8086.) Завершим на этом введение и перейдем к обсуждению возможностей микропроцессора 8088.

1. 3. ОБЩИЕ СВЕДЕНИЯ О МИКРОПРОЦЕССОРЕ 8088

Внутри микропроцессора 8088 информация содержится в группе 16-битовых элементов, называемых *регистрами*. Всего он имеет 14 регистров: 12 регистров данных и адресов и в дополнение к ним указатель команд (регистр адреса команд) и регистр состояния (регистр *флагов*). Можно подразделить 12 регистров данных и адресов на три группы по четыре регистра, а именно на *регистры данных, регистры указателей и индексов и регистры сегментов*.

АДРЕСАЦИЯ

Так как у микропроцессора 8088 указатель команд и адресные регистры имеют по 16 битов, то можно подумать, что он способен адресоваться к памяти объемом не более 64 Кбайт (65 536 байт), т. е. имеет стандартный для 8-битовых микропроцессоров диапазон адресов. Однако на самом деле микропроцессор 8088 всегда генерирует 20-битовые адреса. Он делает это за счет добавления 16-битового смещения к содержимому регистра сегмента, умноженному на 16. Таким образом,

физический адрес = смещение + 16 · (регистр сегмента).

В действительности микропроцессор 8088 вместо *умножения* на 16 использует содержимое регистра сегмента так, как если бы оно имело четыре дополнительных нулевых бита (рис. 1. 1). Добавление нулей аналогично умножению, поскольку при каждом сдвиге на одну позицию влево двоичное число удваивается. Следовательно, перемещение значения регистра сегмента на четыре позиции влево умножает его на 16: 2 · 2 · 2 · 2 = 16.

Например, если смещение адреса имеет значение 10H, где суффикс H обозначает шестнадцатеричное число, а регистр сегмента содержит 2000H, то микропроцессор 8088 произведет вычисление физического адреса следующим образом (операнды показаны в двоичном виде):

```
   0000 0000 0001 0000  (смещение)
+ 0010 0000 0000 0000 0000  (номер блока)
  0010 0000 0000 0001 0000  (физический адрес).
```

Следовательно, мы получим 20-битовый адрес ячейки памяти 20010H.

Имея в своем распоряжении 20-битовый адрес, микропроцессор 8088 может получить доступ к любому из 1 048 576 байт. (Данное значение называется "мегабайт" (Мбайт); 1 Мбайт = 1024 Кбайт.) Это в 16 раз превышает диапазон адресов микропроцессора 8088!

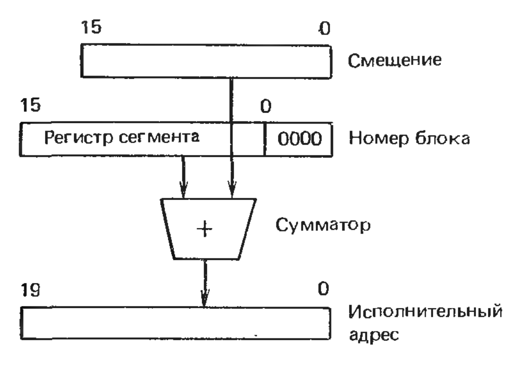


Рис. 1.1. Генерация 20-битового адреса

СЕГМЕНТЫ И СМЕЩЕНИЯ

У большинства микропроцессоров адрес ячейки памяти представляет собой одно число. А у микропроцессора 8088 адрес каждой ячейки памяти задается двумя числами: *номером блока* и *смещением*. Такой странный метод адресации обусловлен тем, что команды программы для микропроцессора 8088 и ее данные должны располагаться в разных частях памяти, другими словами, в разных сегментах. Если, например, Вам требуется адресоваться к ячейке с данными, то микропроцессору 8088 понадобится адрес блока памяти, с которого начинается сегмент данных (из регистра сегмента данных), и позиция желаемой ячейки в этом сегменте (ее смещение). Вспомните, как ищут в городе чей-нибудь дом: сначала находят улицу (считайте ее сегментом), а затем дом с нужным номером (смещением) на этой улице.

К счастью, Вам достаточно задавать только смещение, а номер блока микропроцессор 8088 выберет сам. Несколько позже мы обсудим это обстоятельство более детально.

ВОЗМОЖНОСТИ ПРОГРАММИРОВАНИЯ

Микропроцессор 8088 обладает впечатляющими возможностями программирования. Они пришлись особенно по душе тем программистам, которым ранее приходилось "бороться" с 8-битовыми микропроцессорами. Действительно, микропроцессор 8088 может выполнять арифметические операции над двоичными числами со знаком и без знака, длиной в 8 или 16 битов, а также над десятичными числами, хранящимися как в *упакованном* (по две цифры в байте), так и в *неупакованном* (по одной цифре в байте) виде. Он может также оперировать строками (например, сообщениями) длиной до 64 Кбайт.

Система команд микропроцессора 8088 содержит 92 основных типа команд и предусматривает семь различных способов адресации или режимов доступа к данным. Комбинации типов команд, режимов адресации (имеющие множество комбинаций операндов) и различных типов данных, которые мы только что упоминали, образуют тысячи команд, которые могут быть выполнены микропроцессором 8088. Действительно, сочетание всех этих свойств позволяет микропроцессору 8088 обеспечивать вдвое большую производительность по сравнению с 8085, если оба они работают с одинаковой скоростью.

МЕРА СКОРОСТИ

Как и электронные часы, микропроцессоры управляются кварцевым генератором. Кварцевый генератор испускает импульсы со стабильной фиксированной частотой, задающей скорость работы микропроцессора. В персональных ЭВМ PC и PC/XT кварцевый генератор испускает 4 770 000 импульсов в секунду.

Специалисты по вычислительной технике пользуются другой единицей измерения: числом *колебаний в секунду,* или *герцами*. Импульсы в секунду, колебания в секунду и герцы означают одно и то же, но в этой книге мы будем использовать термин "герц". Таким образом, персональная ЭВМ PC имеет тактовый генератор с частотой 4 770 000 Гц (4,77 МГц).

При частоте 4,77 МГц (4 770 000 колебаний в секунду) тактовый генератор персональной ЭВМ PC "тикает" каждые 210 нс. Мы будем называть это значение циклом тактового генератора ЭВМ.

Самая быстрая команда (например, копирование содержимого одного регистра в другой регистр) выполняется за два цикла, или за 420 нс. Самая медленная

команда, деление двух 16-битовых чисел со знаком, выполняется за 206 циклов тактового генератора, или примерно за 43 мкс. Как видите, даже самая медленная команда выполняется всего за 0, 000043 с!

ОБЛАСТЬ ПОРТОВ ВВОДА-ВЫВОДА

В дополнение к области памяти объемом в 1 Мбайт микропроцессор 8088 может адресоваться к внешним устройствам через 65 536 (64 К) *портов* ввода-вывода. Он имеет специальные команды ввода-вывода, которые позволяют Вам иметь непосредственный доступ к первым 256 портам (от 0 до 255). Другие команды позволяют Вам получить *косвенный* доступ к порту с помощью занесения идентифицирующего его номера (от 0 до 65 535) в регистр данных. Подобно ячейкам памяти любой порт может быть 8- или 16-битовым.

РАСПРЕДЕЛЕНИЕ ПАМЯТИ

Большая часть адресного пространства в 1 Мбайт доступна для системных и прикладных программ, но некоторые ячейки с самыми старшими и самыми младшими адресами используются микропроцессором 8088 для специальных целей. Шестнадцать старших байтов памяти отводятся под команды начальной загрузки системы, которые используются микропроцессором 8088 в момент включения питания. Первые 1024 ячейки отведены под адреса программ, которые исполняются микропроцессором 8088 при прерывании его работы внешним устройством.

ПРЕРЫВАНИЯ

Время от времени всем нам приходится сталкиваться с прерываниями. Одни прерывания нам приятны, другие неприятны, третьи безразличны. При желании Вы можете игнорировать некоторые прерывания, не обращая внимания, скажем, на телефонный или дверной звонок или на ребенка, дергающего Вас за рукав. (Да нет, игнорировать ребенка практически невозможно!) На прерывания других видов Вы просто обязаны реагировать; например, если Вы прокололи шину на железнодорожном переезде, то должны немедленно что-то предпринять.

Прерывания по сути являются требованиями к Вам обратить на них внимание. Аналогично периферийные устройства вычислительной системы могут потребовать, чтобы процессор "обратил на них внимание". Поэтому событие, которое заставляет процессор приостановить выполнение своей программы для выполнения некоторой затребованной деятельности, называется *прерыванием*.

Прерывания существенно увеличивают эффективность вычислительной системы, поскольку они позволяют внешним устройствам "обращать на себя внимание" процессора только по мере надобности. Если бы в системе не было прерываний, то процессору пришлось бы периодически проверять, не требует ли обслуживания какое-нибудь устройство системы. Это похоже на телефон без звонка: пользуясь им, Вам приходилось бы очень часто снимать трубку и проверять, не пытается ли кто-нибудь в Вами соединиться?

ТИПЫ ПРЕРЫВАНИЙ

Микропроцессор 8088 может обрабатывать прерывания двух видов: одни он может игнорировать, а другие обязан обслужить как можно скорее. Прерывания могут быть инициированы внешними устройствами, например дисководами и другими высокоскоростными периферийными устройствами; они могут быть

также инициированы специальными командами или, в определенных ситуациях, самим микропроцессором 8088.

Микропроцессор 8088 может распознать 256 различных прерываний. Каждому из них однозначно соответствует *код типа*, по которому микропроцессор идентифицирует прерывание. Он использует этот код (целое число от 0 до 255) в качестве указателя ячейки, находящейся в области памяти с младшими адресами. Эта ячейка содержит адрес программы обработки данного прерывания, называемый *вектором прерывания*.

Некоторые из этих 256 прерываний используются системным программным обеспечением, поставляемым фирмой IBM к своим персональным ЭВМ IBM PC, а именно резидентной частью операционной системы (называемой BIOS), дисковой операционной системой DOS и интерпретатором языка Бейсик. Остальные прерывания можно использовать по своему усмотрению. Более детально прерывания обсуждаются в гл. 6.

АДРЕСНАЯ ШИНА И ШИНА ДАННЫХ

Микропроцессор 8088 выполнен в виде интегральной микросхемы с 40 выводами, 20 из которых служат для вывода адресов ячеек памяти и образуют так называемую *адресную шину*. Первые 8 проводников адресной шины используются также для ввода данных в микропроцессор и вывода данных из него; они образуют *8-битовую шину данных* микропроцессора 8088. Такое совмещение называется *мультиплексированием* шины данных и адресной шины. (Старшие четыре проводника адресной шины также мультиплексированы: по ним в микропроцессор поступает информация о состоянии операций с памятью и устройствами ввода-вывода.)

1. 4. ВНУТРЕННИЕ РЕГИСТРЫ

Так как эта книга посвящена в первую очередь программированию микропроцессора 8088, то логичнее всего начать эту тему с обсуждения внутренних регистров микропроцессора, доступных для использования. На рис. 1. 2 показаны три группы регистров данных и адресов, 16-битовый указатель команд IP (Instruction Pointer) и 16-битовый регистр флагов.

РЕГИСТРЫ ДАННЫХ

В зависимости от того, чем Вы оперируете: 16-битовыми словами или 8-битовыми байтами, регистры данных можно рассматривать как четыре 16-битовых или восемь 8-битовых регистров. В первом случае регистры имеют имена AX, BX, CX, DX. Эти регистры образованы из 8-битовых регистров AL, AH, BL, BH, CL, CH, DL и DH. Здесь L и H означают младшие (low-order) и старшие (high-order) байты 16-битовых регистров. Например, регистры AL и AH образуют соответственно младший и старший байты регистра AX.

Всеми этими регистрами можно пользоваться при программировании, но следует учитывать, что ряд команд использует их неявным образом, в частности

регистр AX, аккумулятор (accumulator), используется при умножении и делении слов, в операциях ввода-вывода и в некоторых операциях над строками;
регистр AL используется при выполнении аналогичных операций над байтами, а также при преобразовании десятичных чисел и выполнении над ними арифметических операций;
регистр AH используется при умножении и делении байтов;

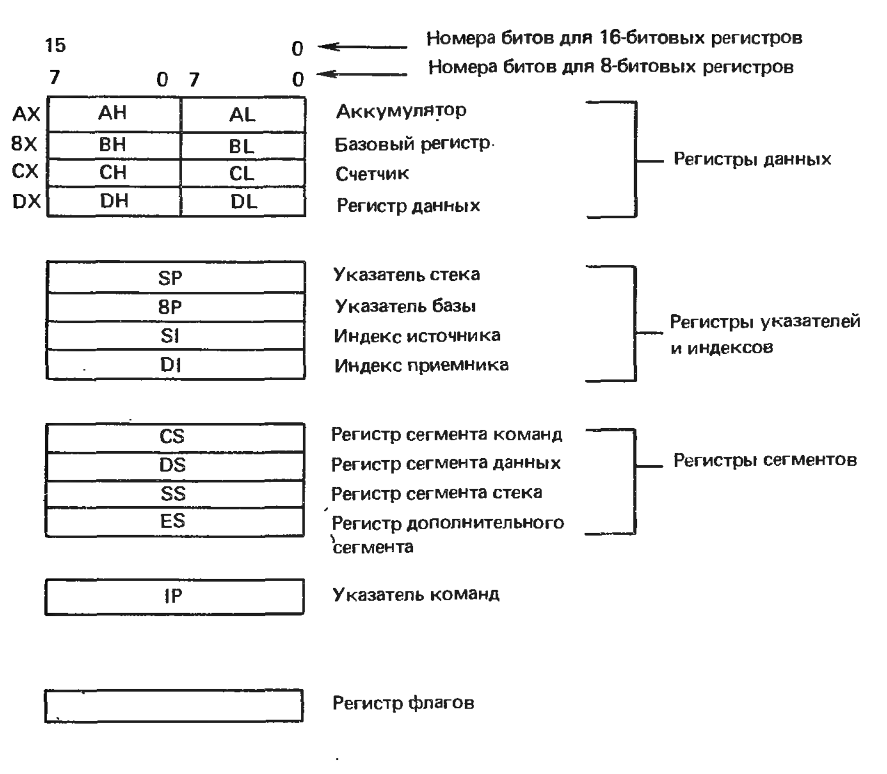


Рис. 1.2. Регистры микропроцессора 8088

регистр BX, базовый регистр (base register), часто используется при адресации данных в памяти;

регистр CX, счетчик (count register), используется как счетчик числа повторений цикла и в качестве номера позиции элемента данных при операциях над строками. *Регистр CL* используется как счетчик при операциях сдвига и циклического сдвига на несколько битов;

регистр DX, регистр данных (data register), используется при умножении и делении слов. Кроме того, в операциях ввода-вывода он используется как номер порта.

Программисты, имеющие опыт работы с микропроцессорами 8080 и 8085, должны заметить, что AH – единственный регистр микропроцессора 8088, не имеющий аналога в этих микропроцессорах. Остальные регистры использовались в них под другими именами; так, в микропроцессоре 8080 регистру AL соответствует регистр A, а регистрам BX, CX и DX соответствуют регистры HL, BC и DE.

Только регистры данных могут использоваться и как 16-битовые, и как 8-битовые. Все регистры остальных групп – 16-битовые.

РЕГИСТРЫ СЕГМЕНТОВ

Ранее уже говорилось, что в ЭВМ, сконструированных на базе микропроцессоров 8086 и 8088, программы и данные хранятся в отдельных областях памяти. Эти области (или *сегменты*) могут иметь объем до 64 Кбайт. Микропроцессор 8088 может иметь дело одновременно с четырьмя сегментами. Начальные адре-

са этих сегментов содержатся в его четырех *регистрах сегментов*. Эти регистры выполняют следующие функции:

регистр сегмента команд CS (code segment) указывает на сегмент, содержащий текущую исполняемую программу. Для вычисления адреса следующей исполняемой команды микропроцессор 8088 добавляет к содержимому регистра CS (умноженному на 16) содержимое указателя команд IP;
регистр сегмента стека SS (stack segment) указывает на текущий сегмент стека. Стек представляет собой область памяти, используемую для временного хранения данных и адресов. Микропроцессор 8088 использует стек для хранения адреса возврата из текущей подпрограммы, но стек можно использовать также для восстановления содержимого регистров, изменяемых при работе программы. Позже мы рассмотрим стеки более подробно;
регистр сегмента данных DS (data segment) указывает на текущий сегмент данных, обычно содержащий используемые в программе переменные;
регистр дополнительного сегмента ES (extra segment) указывает на текущий *дополнительный* сегмент, который используется при выполнении операций над строками.

РЕГИСТРЫ УКАЗАТЕЛЕЙ И ИНДЕКСОВ

Вспомните, что для вычисления адреса команды в сегменте команд микропроцессор 8088 извлекает номер блока памяти из регистра CS, а смещение – из регистра IP. Подобным образом за счет выбора номера блока из соответствующего регистра сегмента, а смещения – из другого регистра осуществляется доступ к данным других сегментов. Для доступа к сегменту данных микропроцессор 8088 извлекает номер блока из регистра DS, а смещение – из регистра BX или *индексного регистра (SI или DI)*. Для доступа к сегменту стека микропроцессор 8088 извлекает номер блока из регистра SS, а смещение – из *регистра указателя (SP или BP)*. Выбирая номер блока из регистра ES, микропроцессор может также получить доступ к дополнительному сегменту (подробнее об этом см. в гл. 2).

УКАЗАТЕЛЬ КОМАНД

Большинство микропроцессоров выполняют программу следующим образом: извлекают из памяти очередную команду, исполняют ее, затем извлекают следующую команду и т. д. Этот подход (при котором цепочка действий выполняется последовательно), естественно, приводит к простоям, так как микропроцессор должен перед исполнением команды подождать ее извлечения из памяти. В микропроцессоре 8088 большая часть этих простоев исключается за счет того, что эти две задачи – извлечение команды и ее исполнение – выполняются отдельными специализированными компонентами микросхемы.

Одна из них, *интерфейс шины* (Bus Interface Unit), извлекает команды из памяти и осуществляет передачу данных между исполнительными компонентами и "внешним миром". Другая компонента, *операционный блок* (Execution Unit), только исполняет команды. Так как эти компоненты независимы, то интерфейс шины может извлекать новую команду из памяти в то самое время, когда операционный блок исполняет ранее извлеченную команду.

Как только интерфейс шины извлекает команду, он передает ее на *конвейер* микропроцессора (электронный эквивалент автомата для продажи сигарет). Таким образом, когда операционный блок заканчивает исполнение команды программы, то обычно может "взять" следующую команду прямо с конвейера.

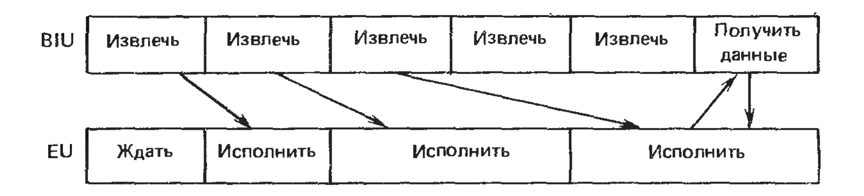


Рис. 1.3. Параллельное выполнение операций на конвейере микропроцессора 8088

Так как интерфейс шины не знает порядка исполнения команд программы, то он всегда извлекает команды с последовательно возрастающими адресами. Поэтому операционному блоку приходится ждать извлечения команды из памяти только в тех случаях, когда в программе управление передается не следующей команде, а какой-то другой. Тогда операционный блок ожидает, пока интерфейс шины не освободит конвейер и не извлечет требуемую команду. Тогда и только тогда микропроцессор 8088 ждет подобно многим другим микропроцессорам извлечения каждой команды. На рис. 1.3 показано параллельное извлечение и исполнение команд в микропроцессоре 8088.

Так как микропроцессор 8088 работает столь необычным образом, то специалисты фирмы Intel подчеркнули отличие своего регистра "следующего исполняемого адреса" от регистров "следующего извлекаемого адреса" других производителей микропроцессоров, назвав его *указателем команд* (IP – instruction pointer) вместо обычного счетчика команд (PC – program counter). Так как регистр IP имеет столь специфическое назначение, то Вы не можете выполнять арифметические операции над его содержимым. Однако микропроцессор 8088 имеет команды, одни из которых изменяют содержимое регистра IP, а другие помещают его в стек и извлекают обратно.

ФЛАГИ

В программе нередко требуется принять решение на основании результата только что исполненной микропроцессором 8088 команды. Например Вам может понадобиться выполнить одни действия, если результат сложения оказался нулем (например, напечатать "Баланс равен нулю!" в программе для бухгалтерских расчетов), и совсем другие действия в противном случае.

В 16-битовом *регистре флагов* фиксируется информация о текущем состоянии дел, которая может помочь Вашей программе принять решение. Шесть битов регистра служат для хранения состояний, а три других могут быть использованы для программного управления режимом работы микропроцессора 8088.

На рис. 1.4 показано, как эти девять флагов размещены в регистре флагов. Флаги имеют следующие значения:

1. *Бит 0, флаг переноса CF* (carry flag), равен 1, если произошел перенос единицы при сложении или заем единицы при вычитании. В противном случае он равен нулю. Кроме того, CF содержит значение бита, который при сдвиге или циклическом сдвиге регистра или ячейки памяти вышел за их границы, и отражает результат операции сравнения. Наконец, CF служит индикатором результата умножения; детали см. в описании бита 11 (OF).
2. *Бит 2, флаг четности PF* (parity flag), равен 1, если в результате операции получено число с четным числом единиц в его битах. В противном случае он равен нулю. Флаг PF в основном используется в операциях обмена данными.

3. *Бит 4, вспомогательный флаг переноса AF* (auxiliary carry flag), аналогичен флагу CF, только контролирует перенос или заем для третьего бита данных. Полезен при выполнении операций над упакованными десятичными числами.

4. *Бит 6, флаг нуля ZF* (zero flag), равен 1, если в результате операции получен нуль; ненулевой результат сбрасывает ZF в нуль.

5. *Бит 7, флаг знака SF* (sign flag), имеет значение только при операциях над числами со знаком. Флаг SF равен 1, если в результате арифметической или логической операции, сдвига или циклического сдвига получено отрицательное число. В противном случае он равен нулю. Иными словами, SF дублирует старший (знаковый) бит результата независимо от того, имеет результат длину 8 или 16 битов.

6. *Бит 8, флаг трассировки TF* (trap flag), разрешает микропроцессору 8088 исполнять програму "по шагам" и используется при отладке программ.

7. *Бит 9, флаг прерывания IF* (interrupt enable flag), разрешает микропроцессору 8088 реагировать на прерывания от внешних устройств. Сбрасывание IF в нуль заставляет микропроцессор 8088 игнорировать прерывания до тех пор, пока IF не станет равным 1.

8. *Бит 10, флаг направления DF* (direction flag), заставляет микропроцессор 8088 уменьшать на единицу (DF = 1) или увеличивать на единицу (DF = 0) регистр(ы) индекса после выполнения команды для работы со строками. Если DF = 0, то микропроцессор 8088 будет обрабатывать строку "слева направо" (от младших адресов к старшим). Если DF = 1, то обработка пойдет в обратном направлении (от старших адресов к младшим или справа налево).

9. *Бит 11, флаг переполнения OF* (overflow flag), в первую очередь служит индикатором ошибки при выполнении операций над числами со знаком. Флаг OF равен 1, если результат сложения двух чисел с одинаковым знаком или результат вычитания двух чисел с противоположными знаками выйдет за пределы допустимого диапазона значений операндов. В противном случае он равен 0. Кроме того, OF = 1, если старший (знаковый) бит операнда изменился в результате операции арифметического сдвига. В противном случае он равен 0.

В сочетании с флагом CF флаг OF указывает длину результата умножения. Если старшая половина произведения отлична от нуля, то OF и CF равны 1; в противном случае оба эти флага равны 0.

Наконец, OF = 0, если частное от деления двух чисел переполняет результирующий регистр.

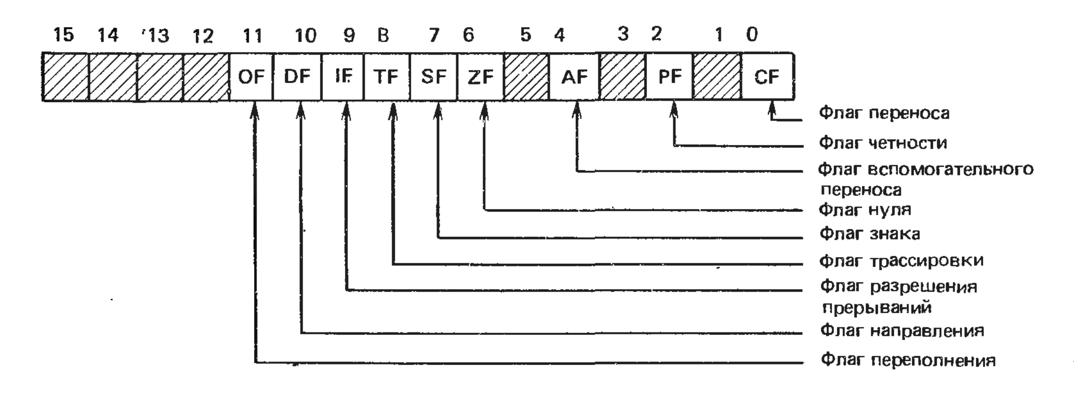


Рис. 1.4. Регистр флагов

На рис. 1.4 заштрихованы позиции неиспользуемых битов (1, 3, 5 и 12 – 15). Когда бы Вы ни прочитали значение регистра флагов, эти биты будут нулевыми.

Не делайте ошибок, рассчитывая, что в кождый момент времени флаги должны быть в определенном состоянии. У микропроцессора 8088 имеются команды для установки или сброса флагов. Если состояние флагов может быть неоднозначным, используйте эти команды.

У микропроцессора 8088 есть команды условной передачи управления, которые проверяют состояния флагов переноса CF, четности PF, нуля ZF, знака SF и переполнения OF. В зависимости от результата проверки выполнение программы продолжается либо со следующей команды, либо с некоторой другой ячейки памяти.

УПРАЖНЕНИЯ

1. Чем различаются системы команд микропроцессоров 8088 и 8086?
2. Какой физический адрес генерирует микропроцессор 8088, если значение смещения 2H сочетается с содержимым регистра сегмента, равным 4000H?
3. Если регистр AX содержит 1A2BH, то что содержит регистр AL?
4. С помощью какого регистра сегмента в основном осуществляется доступ к переменным Вашей программы?
5. По какому биту регистра флагов можно установить, что предшествующее вычитание привело к отрицательному результату?

ГЛАВА 2. ПОЛЬЗОВАНИЕ АССЕМБЛЕРОМ

2.1. ЧТО ТАКОЕ АССЕМБЛЕР?

Язык ассемблера имеет два достоинства: с одной стороны, он позволяет Вам писать программы на уровне команд микропроцессора, а с другой стороны, не требует, чтобы Вы помнили множество числовых кодов. Вместо этого Вы записываете команды, образованные сокращением английских слов, а затем вызываете программу *Ассемблер*, которая преобразует эти сокращения в их числовые эквиваленты.

Программу, написанную в терминах сокращений, будем называть *исходной программой*, а ее форму в числовых кодах команд микропроцессора – *объектной программой*. Таким образом функцией Ассемблера является преобразование исходной программы, доступной Вашему пониманию, в объектную программу, понятную *микропроцессору*.

МАКРОАССЕМБЛЕР ФИРМЫ IBM

Мы не будем описывать все разновидности Ассемблеров, имеющиеся в продаже, а ограничимся одним из наиболее популярных пакетов программ –*Макроассемблером* фирмы IBM. По своим функциям он близок к любому Ассемблеру, который мог оказаться в Вашем распоряжении. В настоящее время фирма IBM выпускает две версии Макроассемблера – 1 и 2. Эти версии похожи, но версия 2 поддерживает составные имена файлов операционной системы DOS 2, воспринимает команды математического сопроцессора 8087 и микропроцессора 80286 фирмы Intel (используемого в персональной ЭВМ IBM PC AT) и предоставляет несколько других возможностей, которые мы рассмотрим позже.

Диск с Макроассемблером содержит две отдельные программы, называемые *Малый ассемблер (ASM)* и *Макроассемблер (MASM)*. Первой программе требуется

всего 64 Кбайт памяти (для версии 2 – 96 Кбайт), второй – 96 Кбайт (для версии 2 под управлением DOS 2 требуется 128 Кбайт). Основное различие между ними состоит в том, что Макроассемблер позволяет Вам задавать свои *макроопределения* (или *макросы*) – группы команд, для вставки которых в программу Вам достаточно указать только их имя, а Малый ассемблер не дает такой возможности. Кроме того, Макроассемблер выдает удобные для чтения сообщения об ошибках, а Малый ассемблер – только коды ошибок. Наконец, Макроассемблер воспринимает команды микропроцессоров 8087 и 80286, а Малый ассемблер – нет.

Так как большинство современных моделей персональных ЭВМ IBM PC имеет по меньшей мере 128 Кбайт памяти, то мы сконцентрируем внимание на Макроассемблере, но будем отмечать его отличия от Малого ассемблера. В фирменном руководстве по Макроассемблеру детально описаны обе программы, поэтому мы не будем претендовать на полноту изложения, а коснемся наиболее часто используемых возможностей и дополним изложение некоторыми таблицами.

2.2. РАЗРАБОТКА ПРОГРАММЫ НА ЯЗЫКЕ АССЕМБЛЕРА

Хотя внешне программы на языке ассемблера сильно отличаются от программ на языке Бейсик, тем не менее разрабатываются они одинаково. Однако разработка программ на языке ассемблера требует большей *техники*. В ней можно выделить шесть этапов:

1. Поставьте задачу и составьте проект программы. На этом этапе нередко составляют *блок-схемы* – эскиз выполняемых программой действий.
2. Введите команды программы в ЭВМ с помощью *редактора*. Лучше всего сначала ввести комментарии на обычном языке с описанием предполагаемых действий, а затем вставить между ними соответствующие команды.
3. Оттранслируйте программу с помощью *Ассемблера*. Если Ассемблер обнаружит ошибки, исправьте их редактором и оттранслируйте программу заново.
4. Преобразуйте результат работы Ассемблера в исполняемый модуль с помощью *загрузчика*.
5. Исполните (вызовите) программу.
6. Проверьте результаты. Если они не соответствуют Вашим ожиданиям, надо найти ошибки, другими словами, отладить программу.

Если программа проста и коротка, то эти этапы не отнимут у Вас много времени. Однако более сложные и длинные программы потребуют больше времени на каждом этапе и в первую очередь на этапе проектирования. В конце этой главы в разделе "Разработка программы методом сверху вниз" мы обсудим эффективный подход к проектированию программ.

РЕДАКТОР

Редактор, упомянутый на втором этапе, представляет собой программу, которая позволяет Вам вводить и подготавливать собственные программы. В качестве редактора можно использовать любую популярную программу редактирования или обработки текстов, которая формирует "чистый" текст в кодах ASCII (не содержащий управляющих кодов описания формата). К таким программам относятся WordStar, Microsoft Word, MultiMate и Personal Editor фирмы IBM.

Если ни одной из них у Вас нет, то можно воспользоваться программой EDLIN, которая поставляется на диске с операционной системой DOS. Эта программа представляет собой построчный редактор; другими словами, его команды позволяют производить действия над пронумерованными строками Вашей программы. В разд 2.7 мы расскажем, как ей пользоваться.

АССЕМБЛЕР

ЭВМ не может непосредственно воспринять программу, которую Вы подготовили редактором. Ассемблер должен преобразовать ее в *объектную программу*, понятную ЭВМ.

ЗАГРУЗЧИК LINK

Операционная система DOS фирмы IBM может хранить программу в любом подходящем месте памяти и освобождает Вас от необходимости думать, куда ее поместить. Но, чтобы воспользоваться этим, Вам надо преобразовать оттранслированную программу к виду, позволяющему перемещение (в литературе по ЭВМ используется термин "перемещаемая программа"). Эта программа с помощью загрузчика – программы LINK, которая поставляется на диске с дополнительными программами DOS. Следует учесть, что Макроассемблер версии 2 включает в себя расширенную версию загрузчика (2.20) для пользователей, имеющих DOS 2.0 и более поздние версии. Эта версия стандартно поставляется с DOS версии 3.0.

В фирменном руководстве *объектным модулем* называется дисковый файл, содержащий результат трансляции Ассемблером. А файл, содержащий перемещаемую версию оттранслированной программы, называется *исполняемым модулем*. Таким образом, функцией загрузчика LINK является создание исполняемого модуля из объектного модуля.

ДРУГАЯ ФУНКЦИЯ ЗАГРУЗЧИКА

Вы можете составлять программу как авиамодель, из отдельных частей. Для этого введите в ЭВМ часть программы и оттранслируйте ее Ассемблером. Если он выдаст сообщения об ошибках, исправьте их и оттранслируйте программу заново. После того, как этот первый модуль будет оттранслирован без ошибок, повторите данный процесс для второго модуля, третьего и т. д. В конце концов у Вас будет несколько объектных модулей, которые в совокупности должны выполнять все то, что требуется от Вашей программы.

По-видимому, Вы уже догадались, в чем состоит другая функция загрузчика: он создает один исполняемый модуль из нескольких оттранслированных объектных модулей.

Вы должны вызывать загрузчик для любой написанной Вами программы, даже если она состоит только из одного объектного модуля. Одномодульные программы загрузчик сразу преобразует в перемещаемый модуль. Если программа состоит из двух или более модулей, то загрузчик сначала объединяет их, а затем преобразует результат в перемещаемый модуль.

ОТЛАДЧИК DEBUG

Завершенную программу можно вызвать для исполнения двумя способами:

1. Набрать ее имя в качестве команды DOS.
2. Выполнить ее под управлением программы DEBUG.

Обычно программу следует вызывать из операционной системы DOS только в том случае, если Вы уверены в ее безошибочной работе. Пока она не будет отлажена, вызывайте ее под управлением отладчика DEBUG.

Отладчик DEBUG (он находится на диске с дополнительными программами DOS) позволяет управлять процессом исполнения программы. Наряду с другими функциями DEBUG позволяет изображать и изменять значения переменных, останавливать исполнение программы в заданной точке или исполнять программу по шагам. Тем самым DEBUG служит Вам инструментом для поиска и исправления ошибок в программе.

РАЗРАБОТКА ПРОГРАММЫ МЕТОДОМ "СВЕРХУ ВНИЗ"

При создании программы для ЭВМ естественным побуждением является начать действовать так, как будто Вы имеете дело с карандашом и бумагой: последовательно ввести первую команду, затем вторую, третью и так далее до самого конца. Этот "силовой" метод может быть пригоден в случае простой или короткой программы, но обычно он приводит к ошибкам и к программам, которые трудно понять, а впоследствии еще труднее модифицировать. Благодаря удобным возможностям редактирования текстов, предоставляемым программами обработки текстов и редакторами, можно воспользоваться более легким, удобным и эффективным методом разработки программ. Он называется *методом разработки "сверху вниз"*.

Попросту говоря, разработка методом "сверху вниз" означает, что вначале составляется набросок программы в виде текста на обычном языке, а затем этот набросок постепенно дополняется деталями. Набросок должен представлять собой ряд строк, в которых описаны действия программы. Например, при разработке программы, которая выполняет одну из нескольких функций по выбору пользователя, набросок может выглядеть следующим образом:

```
; Изобразить меню возможных функций
; Запросить у пользователя выбор из меню
; Прочитать ответ пользователя
; Проверить допустимость ответа
; Если ответ допустим, выполнить требуемую функцию
```

Точки с запятой означают, что эти строки представляют собой не команды, а комментарии. Они играют ту же роль, что и операторы REM в Бейсике.

Исходя из этого текста можете с помощью редактора вставить между этими строками команды. Так как каждая строка описывает относительно небольшую задачу, проще всего выполнить каждую задачу в отдельности, проверяя решение перед тем, как двинуться дальше. Иначе говоря, начните со вставки первой группы команд (в нашем примере с команд для изображения меню), затем запомните полученную программу на диске и выполните все последующие этапы (трансляцию, загрузку и исполнение). Выполнение такой частичной программы покажет, правильно ли она работает. Если она работает неправильно, отладьте ее и попробуйте еще раз. После того, как первая часть отлажена, перейдите ко второй части, затем к третьей и т. д.

Может показаться, что это ужасно медленный метод разработки программы, но он имеет несколько преимуществ:

1. Дисциплинирует разработку программы.

2. Комментарии представляют собой род документации высокого уровня для законченного продукта.

3. Обеспечивает правильность работы каждой части до того, как Вы перейдете к разработке следующей.

2.3. ОПЕРАТОРЫ ИСХОДНОЙ ПРОГРАММЫ

Обсудив метод разработки программ, посмотрим, из чего же они состоят. Программа, которую Вы пишете (исходная программа), представляет собой последовательность *операторов*, описывающих выполняемые операции. Оператором (строкой) исходной программы может быть или команда, или псевдооператор языка ассемблера.

Команды языка ассемблера представляют собой краткую нотацию системы команд микропроцессора 8088. В некоторых руководствах они называются *машинными командами*, потому что они сообщают *машине* (микропроцессору 8088), что ей делать. В отличие от них *псевдооператоры* сообщают *Ассемблеру*, что ему делать с командами и данными, которые Вы вводите.

Операторы любого вида могут включать в себя *операции*, которые дают Ассемблеру информацию об операндах, относительно которых нет полной определенности. Ниже мы обсудим команды языка ассемблера, псевдооператоры и операции.

КОНСТАНТЫ В ОПЕРАТОРАХ ИСХОДНОЙ ПРОГРАММЫ

Ассемблер позволяет вводить константы в различной форме. Наиболее распространены следующие виды констант:

1. *Двоичная* – последовательность из цифр 0 и 1, заканчивающаяся буквой B; например 10111010B.
2. *Десятичная* – последовательность из цифр от 0 до 9, которая может заканчиваться буквой D; например 129D или 129.
3. *Шестнадцатеричная* – последовательность цифр от 0 до 9 и букв от A до F, заканчивающаяся буквой H. Первым символом должна быть одна из цифр от 0 до 9; например 0E23H. (В данном случае 0 информирует Ассемблер о том, что E23H – число, а не идентификатор или имя переменной.)
4. *Литерал* – строка букв, цифр и других символов, заключенная в кавычки или апострофы. В языке ассемблера фирмы IBM предусмотрены две формы потому, что иногда приходится вставлять кавычки или апострофы в текст сообщения, например "Ваш ответ на запрос 'Попробуйте снова' ошибочен".

ОТРИЦАТЕЛЬНЫЕ ЧИСЛА

Вы можете вводить и отрицательные числа. В случае десятичного числа перед ним достаточно поставить знак минус (например, –32). Если число двоичное или шестнадцатеричное, то его надо вводить в дополнительном коде. Например 11100000B и 0E0H – формы записи десятичного числа –32 в дополнительном коде.

2.4. КОМАНДЫ ЯЗЫКА АССЕМБЛЕРА

Каждая команда языка ассемблера в исходной программе может иметь до четырех *полей* следующего вида:

```
[Метка:] Мнемокод [Операнд] [;Комментарий]
```

Таким образом, обязательным является только поле мнемонического кода (мнемокода). Поля метки и комментария во всех случаях не обязательны. Поле операнда заполняется только для тех команд, которым *требуется* операнд; в противном случае его можно опустить. (Квадратные скобки вокруг полей метки, операнда и комментария показывают, что эти поля не обязательны; ни в коем случае *не набирайте* эти скобки при вводе программ.)

Вы можете набирать содержимое поля в любом месте строки, но обязательно разделяйте поля хотя бы одним пробелом (или символом табуляции). Примером команды языка ассемблера, где все четыре поля присутствуют, служит команда

```
GETCOUNT: MOV CX,DI      ;Инициализация счетчика
```

ПОЛЕ МЕТКИ

Поле метки служит для присваивания *имени* команде языка ассемблера. По нему на эту команду могут ссылаться другие команды программы. Заметьте, что метки в языке ассемблера играют ту же роль, что и номера строк в Бейсике.

Метка команды может содержать до 31 символа и должна заканчиваться двоеточием (:). В нее могут входить

буквы от A до Z или от a до z (Ассемблер не различает строчные и прописные буквы);
цифры от 0 до 9;
следующие специальные знаки: ? . @ _ $.

Метку можно начать любым символом, кроме цифры, но если Вы используете точку (.), то она обязательно должна быть первым символом метки. Слова AH, AL, AX, BH, BL, BX, BP, CH, CL, CX, CS, DH, DL, DX, DI, DS, ES, SI, SP и ST являются именами регистров и не должны использоваться в качестве меток. Не используйте в качестве меток также имена команд или *мнемокоды* (они перечислены в гл. 3).

В метку нельзя вставлять пробел, его можно заменить символом подчеркивания _. Например, приведенный выше оператор можно переписать в виде

```
GET_COUNT: MOV CX,DI      ;Инициализация счетчика
```

Конечно, GET_COUNT читается удобнее, чем GETCOUNT.

ВЫБОР ИМЕНИ ДЛЯ МЕТОК

Так как Ассемблер позволяет Вам использовать многочисленные комбинации букв, цифр и знаков, то почти любая придуманная Вами метка окажется допустимой. Тем не менее мы дадим рекомендации для выбора меток.

Выбирайте имена покороче, но достаточно понятные. Так, имя MPH предпочтительнее, чем MILES_PER_HOUR, а CUR_YR – разумное сокращение для имени CURRENT_YEAR.

Выбирайте такие имена, чтобы их легче было набрать без ошибок. Обычно затруднен набор нескольких букв подряд (например, HHHH) и близких по начертанию символов (буква O и цифра 0, буква I и цифра 1, буква S и цифра 5). Нет никаких причин увеличивать количество ошибок при наборе – их и так предостаточно.

Не используйте метки, которые можно спутать между собой, например XXXX и XX4X. Незачем искушать судьбу и законы Мэрфи.

ПОЛЕ МНЕМОКОДА

Поле мнемокода содержит имя команды микропроцессора. Имена состоят из двух – шести букв. Например, MOV – имя команды пересылки данных (move – переместить), а ADD – имя команды сложения (add – сложить). Для трансляции каждого мнемокода программы в его числовой эквивалент Ассемблер использует внутреннюю таблицу.

Во многих командах кроме мнемокода надо указывать один или два операнда. Ассемблер по мнемокоду узнает, сколько должно быть операндов и какого типа, а затем обрабатывает поле операнда. Допустимые мнемокоды будут описаны в гл. 3.

ПОЛЕ ОПЕРАНДА

В поле операнда микропроцессору 8088 сообщается, где найти данные, подлежащие обработке. Например, в команде *пересылки*

```
MOV CX,DX
```

указано, что надо скопировать содержимое регистра DX в регистр CX.

Для некоторых команд наличие поля операнда обязательно, для других команд оно запрещено. Если это поле присутствует, то в нем должно быть один-два операнда, отделенных от мнемокода по крайней мере одним пробелом или символом табуляции. Между собой операнды разделяются запятой.

В командах с двумя операндами первый из них представляет собой *приемник*, а второй – *источник*. Операнд-источник определяет значение, которое берется микропроцессором для сложения, вычитания, сравнения со значением операнда-приемника или просто для загрузки в операнд-приемник. (В вышеприведенном примере оператор MOV означает, что значение операнда-источника DX надо запомнить в операнде-приемнике CX.) Поэтому при исполнении команды операнд-источник никогда не изменяется, в то время как операнд-приемник изменяется почти всегда. В гл. 3 мы обсудим виды адресации для каждой команды микропроцессора 8088.

ПОЛЕ КОММЕНТАРИЕВ

Подобно оператору REM в Бейсике это необязательное поле позволяет Вам описывать назначение операторов исходной программы для облегчения ее понимания. Перед комментарием указывается точка с запятой (;), которая должна быть отделена от предыдущего поля по крайней мере одним пробелом или символом табуляции. Ассемблер игнорирует комментарии при трансляции, но сохраняет их в листинге программы.

В комментариях надо описывать не столько действие отдельной команды, сколько ее роль в программе. Например, комментарий

```
MOV CX,0     ;Обнулить регистр счетчика
```

более содержателен, чем комментарий

```
MOV CX,0     ;Загрузить 0 в регистр CX
```

САМОСТОЯТЕЛЬНЫЕ КОММЕНТАРИИ

Вы можете вводить самостоятельные комментарии для описания программы или какой-нибудь ее части. Для этого достаточно начать строку точкой с запятой (;). Встретив точку с запятой в начале строки с комментарием, ассемблер игнорирует остальную часть этой строки.

2.5. ПСЕВДООПЕРАТОРЫ

Псевдооператоры управляют работой Ассемблера, а не микропроцессора. С помощью псевдооператоров можно определять сегменты и процедуры (т. е. подпрограммы), давать имена командам и элементам данных, резервировать рабочие области памяти и выполнять множество других важных "хозяйственных" задач. В отличие от команд языка ассемблера большинство псевдооператоров не генерирует объектного кода.

Псевдооператоры данных могут содержать до четырех полей:

```
[Идентификатор] Псевдооператор [Операнд] [;Комментарий]
```

Таблица 2.1. Наиболее распространенные псевдооператоры

Тип	Псевдооператоры		
Псевдооператоры данных	ASSUME COMMENT DB DW DD	END ENDP ENDS EQU = (знак равенства)	EXTRN INCLUDE PROC PUBLIC SEGMENT
Псевдооператоры управления листингом	PAGE	SUBTTL	TITLE

Как показывают квадратные скобки, обязательным является только поле псевдооператора. Для одних псевдооператоров идентификатор обязателен, для других запрещен, для третьих не обязателен. То же относится и к операнду. Комментарии всегда не обязательны. Как и в случае команд языка ассемблера, поле псевдооператора может начинаться в любом месте строки при условии, что оно отделено от предыдущего поля по крайней мере одним пробелом или символом табуляции.

В Макроассемблере предусмотрено около 60 различных псевдооператоров. В этом разделе мы обсудим наиболее распространенные из них (в разд.2.8 будут рассмотрены некоторые сложные псевдооператоры, а в гл. 9 – псевдооператоры для создания макроопределений). В табл. 2.1 приведены наиболее распространенные псевдооператоры, которые делятся на две группы: псевдооператоры данных и псевдооператоры управления листингом.

Примечание. Не старайтесь запоминать материал этого раздела. Прочтите его бегло, а позже, если Вам понадобятся детали, вернитесь к нему.

ПСЕВДООПЕРАТОРЫ ДАННЫХ

Как показано в табл. 2.2, псевдооператоры данных Ассемблера можно разделить на пять функциональных групп.

Таблица 2.2. Псевдооператоры данных

Псевдооператор	Функция
Определение идентификаторов	
EQU	*Формат*: имя EQU текст *или* имя EQU **числовое_выражение** Постоянно присваивает значение **текст** или **числовое_выражение** идентификатору **имя**
	Формат: имя = **числовое_выражение** Значение **числовое_выражение** присваивается идентификатору **имя**, но может быть переприсвоено

Псевдооператор	Функция
Определение данных	
DB	*Формат*: [имя] DB выражение [, . . .] Определяет переменную или присваивает ячейке памяти начальное значение. Резервирует один или несколько байтов
DW	*Формат*: [имя] DW выражение [, . . .] Аналогичен псевдооператору DB, но резервирует двухбайтовые слова
DD	*Формат*: [имя] DD выражение [, . . .] Резервирует четырехбайтовые двойные слова
Внешние ссылки	
PUBLIC	*Формат*: PUBLIC идентификатор [, . . .] Делает определенный ранее идентификатор доступным другим модулям программы, которые впоследствии должны быть присоединены к данному модулю
EXTRN	*Формат*: EXTRN имя: тип [, . . .] Указывает, что имя определено в другом модуле программы
INCLUDE	*Формат*: INCLUDE файл Вставляет содержимое указанного файла в текущий файл исходной программы
Определение сегмента/процедуры	
SEGMENT	*Формат*: имя_сег SEGMENT [тип_подгонки] [тип_связи] ['класс'] . . . . . . имя_сег ENDS Определяет границы сегмента программы. Каждое определение SEGMENT должно завершаться оператором ENDS
ASSUME	*Формат*: ASSUME регистр_сег: имя_сег [, . . .] или ASSUME регистр_сег: NOTHING [, . . .] Сообщает Ассемблеру, какой регистр сегмента связан с сегментом программы. Оператор ASSUME NOTHING отменяет действие всех предыдущих операторов ASSUME для данного регистра

Псевдооператор	Функция
PROC	*Формат*: имя PROC [NEAR] или имя PROC FAR . . . . . . RET имя ENDP Присваивает имя последовательности операторов. Каждое определение, начинающееся оператором PROC, должно заканчиваться оператором ENDP
Управление трансляцией	
END	*Формат*: END [метка точки входа] Отмечает конец исходной программы

ПСЕВДООПЕРАТОРЫ ОПРЕДЕЛЕНИЯ ИДЕНТИФИКАТОРОВ

Эти псевдооператоры позволяют Вам присвоить *выражению* символическое *имя* (идентификатор). Выражение может быть 16-битовой константой, ссылкой на адрес, другим символическим именем, префиксом сегмента и операндом, меткой команды. После присваивания имени Вы можете пользоваться им всюду, где требуется указать это выражение.

Псевдооператоры EQU (equate – приравнять) и = (знак равенства) сходны по назначению, но различаются следующим:

1. Определенные знаком "=" идентификаторы можно переопределять, а определенные псевдооператором EQU – нельзя.
2. Псевдооператор EQU можно использовать как с числовыми, так и с текстовыми выражениями, а знак "=" только с числовыми.

Псевдооператор EQU удобен для присваивания простых имен числам, сложным комбинациям адресов и другим подобным объектам, которые неоднократно используются в программе. Например:

```
K     EQU 1024          ;Присвоить имя константе
TABLE EQU DS:[BP][SI]   ;Присвоить имя комбинации адресов
SPEED EQU RATE          ;Определить синоним
COUNT EQU CX            ;Присвоить имя регистру
```

Будучи выражением, операнд может содержать простые математические преобразования, которые будут выполнены Ассемблером во время трансляции. Например:

```
DBL_SPEED    EQU 2*SPEED
MINS_PER_DAY EQU 60*24
```

Приведем примеры употребления псевдооператора "=":

```
CONST = 56         ;Аналогично CONST EQU 56, но теперь
CONST = 57         ; CONST можно переопределить явно или
CONST = CONST+1    ; через ее предыдущее определение
```

ПСЕВДООПЕРАТОРЫ ОПРЕДЕЛЕНИЯ ДАННЫХ

Во многих программах ячейки памяти используются для хранения *переменных* – поименованных элементов данных, содержимое которых может быть изменено по мере необходимости. Наиболее употребительными псевдооператорами, резервирующими память для переменных, являются псевдооператоры DB (Define Byte – определить байт), DW (Define Word – определить слово) и DD (Define Double Word – определить двойное слово).

Они различаются в основном по объему резервируемой памяти. Псевдооператор DB резервирует 8-битовые байты, DW – 2-байтовые слова, а DD – 4-байтовые двойные слова. Определяя переменную, Вы можете задать ее начальное значение либо просто зарезервировать для нее место в памяти, а значение присвоить позже.

Псевдооператоры определения данных имеют следующий формат:

```
[имя] DB выражение [,...]
[имя] DW выражение [,...]
[имя] DD выражение [,...]
```

где операнд **выражение** может принимать одну из нескольких форм (в зависимости от Ваших намерений).

Например, выражение может быть *константой*. Следующие операторы показывают максимальные и минимальные допустимые десятичные значения для переменных длиной в байт или слово:

```
BU_MAX DB   255   ;Максимальное значение байта без знака
BS_MAX DB   127   ;Максимальное значение байта со знаком
BS_MIN DB  -128   ;Минимальное значение байта со знаком

WU_MAX DW  65535  ;Максимальное значение слова без знака
WS_MAX DW  32767  ;Максимальное значение слова со знаком
WS_MIN DW -32768  ;Минимальное значение слова со знаком
```

Псевдооператоры можно использовать для создания в памяти таблиц, перечисляя элементы таблицы через запятую. Следующие последовательности задают две таблицы по 12 элементов в каждой, одна из байтов,а другая из слов:

```
B_TABLE DB 0,0,0,0,8,-13                ;Таблица байтов
        DB 100,0,55,63,63,63
W_TABLE DW 1025,567,-30222,0,90,-129 ;Таблица слов
        DW 17,645,26534,367,78,-17
```

Здесь мы разместили элементы таблицы на двух строках по шесть значений в каждой, но в одном псевдооператоре можно указывать *любое* число значений, лишь бы они поместились на строке длиной 132 позиции.

Обратите внимание на то, что первые четыре элемента и последние три элемента таблицы B_TABLE имеют одинаковые значения (соответственно 0 и 63). У Ассемблера есть операция DUP (duplicate – дублировать), позволяющая повторять операнды, не набирая их каждый раз заново. С помощью операции DUP определение таблицы B_TABLE можно записать короче:

```
B_TABLE DB 4 DUP(0),8,-13,-100,0,55,3 DUP(63)
```

При определении переменной без присваивания ей начального значения надо указывать в поле выражения вопросительный знак (?). Например, следующие операторы резервируют соответственно байт и слово памяти, но не заносят в них какое-либо значение:

```
HIGH_TEMP  DB ?
AVG_WEIGHT DW ?
```

Имейте в виду, что знак (?) только *резервирует* память для переменных HIGH_TEMP и AVG_WEIGHT и никаких *начальных значений не задает*. Не надейтесь, что эти переменные будут содержать 0 или какое-либо иное специфическое значение.

Вы можете зарезервировать ячейки памяти для целой таблицы. Например, оператор

```
MONTHLY_SALES DW 12 DUP(?)
```

зарезервирует 12 слов памяти. Он аналогичен оператору Бейсика

```
DIM MONTHLY_SALES%(12).
```

Псевдооператор DB может воспринимать в качестве выражения также *строку символов*. Это позволяет заносить в память сообщения об ошибках, заголовки таблиц и другой текст. Для этого текст надо заключить в кавычки или апострофы. Приведем два примера:

```
POLITE_MSG DB 'Вы ввели слишком большое число'
           DB 'Оно не может быть обработано'
           DB 'Пожалуйста, введите данные заново'
RUDE_MGG DB 'Попробуй снова, тупица!'
```

Переменные используются также для хранения *адресов ячеек памяти*, на которые могут ссылаться команды Вашей программы. Как Вы уже знаете, каждый адрес имеет две компоненты: номер блока и смещение. Если метка находится в том же сегменте, что и команда, которая на нее ссылается, то достаточно указать только смещение. Так как смещение имеет длину 16 битов, то для его хранения надо использовать оператор DW. Например, оператор

```
HERE_NEAR DW HERE
```

присвоит 16-битовому смещению метки HERE имя HERE_NEAR. Содержащую смещение переменную будем называть *указателем*.

Если метка и команда, которая на нее ссылается, находятся в разных сегментах, то кроме смещения микропроцессору 8088 надо знать еще и номер блока метки. Оба этих компонента адреса можно получить с помощью оператора DD. Например, оператор

```
HERE_FAR  DD HERE
```

присвоит 16-битовое смещение и 16-битовый номер блока метки HERE 32-битовой переменной HERE_FAR. Переменную, содержащую оба компонента адреса, будем называть *вектором*.

ПСЕВДООПЕРАТОРЫ ОПРЕДЕЛЕНИЯ СЕГМЕНТА/ПРОЦЕДУРЫ

Псевдооператоры SEGMENT и ENDS делят исходную программу на сегменты. Как мы уже упоминали, в программе может быть до четырех видов сегментов: сегмент данных, сегмент команд, дополнительный сегмент и сегмент стека.

Например, сегмент данных может выглядеть следующим образом:

```
DATASEG   SEGMENT  PARA  PUBLIC  'DATA'
   A        DB  ?
   B        DB  ?
   SQUARES  DB  1,4,9,16,25,36,49,64
DATASEG   ENDS
```

а сегмент команд может иметь следующий вид:

```
PROGCODE   SEGMENT   PARA   PUBLIC 'CODE
             ...
             ...
             MOV   AX,BX
             MOV   CL,DH
             MOV   DI,CX
             ...
             ...
PROGCODE    ENDS
```

Слова SEGMENT и ENDS отмечают начало и конец сегмента. Они не сообщают Ассемблеру, какого рода сегмент должен быть определен. Для этой цели используется отдельный псевдооператор ASSUME.

Оператор ASSUME имеет формат

```
ASSUME регистр_сег:имя_сег[,...]
```

где **регистр_сег** – имя одного из регистров сегмента DS, CS, SS или ES, а **имя_сег** – имя, указанное в псевдооператоре SEGMENT. При этом DS указывает на сегмент данных, CS – на сегмент команд, SS – на сегмент стека, а ES – на дополнительный сегмент.

Оператор ASSUME помогает Ассемблеру преобразовывать метки в адреса, сообщая, каким регистром сегмента Вы хотите воспользоваться при адресации этих меток. Например, оператор

```
ASSUME DS:DSEG
```

указывает Ассемблеру: "Сегмент по имени DSEG – это мой сегмент данных. Когда при трансляции программы будет обнаружено упоминание метки из сегмента DSEG, сообщи микропроцессору, что номер блока метки надо извлечь из регистра DS. Со своей стороны обещаю, что DS укажет на начало DSEG".

Обычно оператор ASSUME помещают сразу же за оператором SEGMENT, определяющим сегмент команд. Так, в предыдущей программе, состоявшей из двух сегментов, сегмент команд должен был бы иметь следующий вид:

```
PROGCODE   SEGMENT    PARA PUBLIC 'CODE'
           ASSUME     CS:PROGCODE,DS:DATASEG
           MOV        AX,DATASEG  ;Установить DS на начало DATASEG
           MOV        DS,AX
           ..
           ..
           MOV        AX,BX
           MOV        CL,DH
           MOV        CX,DI
           ..
           ..
PROGCODE   END
```

Еще раз обратите внимание на то, что мы должны явным образом загрузить адрес начала сегмента данных в регистр DS: командой ASSUME этого сделать нельзя.

Псевдооператоры PROC и ENDP отмечают начало и конец *процедуры*. Процедура представляет собой совокупность команд, которые должны исполняться в разных местах программы. Когда Ваша программа вызывает процедуру, то микропроцессор 8088 исполняет ее, а затем возвращается к тому месту программы, где был сделан вызов. Так как Вы пишете в программе текст процедуры только один раз, то этот прием освобождает Вас от повторения текста всюду, где требуются команды процедуры, и тем самым укорачивает программу.

Каждая процедура должна начинаться оператором PROC и заканчиваться оператором ENDP. Если, кроме того, она содержит команду RET (Return From Procedure – возврат из процедуры), а в большинстве случаев это так, то мы назовем ее *подпрограммой*. Команда RET заставляет микропроцессор 8088 вернуться к тому месту, где была вызвана процедура, и по своему действию аналогична оператору RETURN в Бейсике.

Процедуре всегда приписан один из двух атрибутов *дистанции*: NEAR (близкая процедура) и FAR (далекая процедура). Он должен быть указан в качестве операнда оператора PROC. Если операнд опущен, то подразумевается атрибут NEAR.

Процедура с атрибутом NEAR может быть вызвана только из того сегмента команд, в котором она определена. Например

```
CSEG        SEGMENT    PARA PUBLIC 'CODE'
            ASSUME     CS:CSEG
CALLER        PROC
              ..
              CALL CALLEE    (Вызвать процедуру)
              ..
              RET
CALLER        ENDP
CALLEE        PROC NEAR      (Вызываемая процедура)
              ..
              ..
              RET
CALLEE        ENDP
CSEG        ENDS
```

Здесь вызываемая процедура (CALLEE) определена в том же сегменте, что и команда CALL. Однако такие две процедуры могут находится даже в разных программных модулях (т. е. в разных дисковых файлах), лишь бы их сегменты команд имели одинаковые имена. Связи между модулями обеспечиваются псевдооператорами EXTRN и PUBLIC, которые мы обсудим в следующем разделе.

Процедура с атрибутом FAR может быть вызвана из любого сегмента команд, например

```
CSEG        SEGMENT    PARA PUBLIC 'CODE'
            ASSUME     CS:CSEG
CALLER        PROC
              ..
              CALL CALLEE    (Вызвать процедуру)
              ..
              RET
CALLER        ENDP
CSEG        ENDS

CSEG1       SEGMENT    PARA PUBLIC 'CODE'
            ASSUME     CS:CSEG1
CALLEE        PROC FAR       (Вызываемая процедура)
              ..
              ..
              RET
CALLEE        ENDP
CSEG        ENDS
```

Когда микропроцессор 8088 вызывает процедуру, то он помещает адрес возврата в стек. Этот адрес будет извлечен при выполнении команды RET. Если процедура имеет атрибут NEAR, то при вызове в стек помещается только смещение (содержимое указателя команд IP). Если процедура имеет атрибут FAR, то при вызове в стек помещается и номер блока (содержимое регистра сегмента команд CS), и содержащееся в регистре IP смещение (в указанном порядке).

При трансляции программы Ассемблером каждая команда RET преобразуется в машинную команду, которая указывает микропроцессору 8088, сколько слов с адресами возврата понадобится извлечь из стека. Команда RET в процедуре с атрибутом NEAR заставит микропроцессор извлечь из стека одно слово (содержимое регистра IP); команда RET в процедуре с атрибутом FAR заставит его извлечь два слова (содержимое регистров IP и CS).

Приведем несколько правил, которые помогут решить, какой из атрибутов, NEAR или FAR, надо приписывать Вашей процедуре:

1. Вы можете вызвать программу из операционной системы DOS или из отладчика DEBUG. Но так как и DOS, и DEBUG размещены в сегментах команд, отличных от сегмента команд Вашей программы, то ее *основная процедура должна иметь атрибут FAR.*
2. Если Вы всегда даете сегменту команд одно и то же имя (например, CSEG), то приписывайте всем процедурам, кроме основной, атрибут NEAR.
3. Если Вы используете другие программы, то приписывайте их процедурам атрибут FAR. О том, как это сделать, мы кратко расскажем при обсуждении псевдооператора FAR.

ПСЕВДООПЕРАТОРЫ ВНЕШНИХ ССЫЛОК

Эти псевдооператоры позволяют Вам использовать информацию, находящуюся в других программных модулях или файлах.

Псевдооператор PUBLIC делает указанный в нем идентификатор доступным для других программных модулей, которые впоследствии могут загружаться вместе с данным модулем. Идентификатор может быть именем переменной, меткой (кроме меток оператора PROC) или именем, определенным псевдооператором = или EQU.

Псевдооператор EXTRN описывает идентификаторы, которые определены (и объявлены в операторе PUBLIC) в других программных модулях. Он имеет формат

```
EXTRN имя:тип[,...]
```

где **имя** – идентификатор, определенный в другом программном модуле, а **тип** задается следующим образом:

Если **имя** является идентификатором, определенным в сегменте данных или в дополнительном сегменте, то **тип** может принимать значения BYTE, WORD или DWORD.

Если **имя** – метка процедуры, то **тип** может быть NEAR или FAR.

Если **имя** относится к константе, определенной псевдооператорами = или EQU, то **тип** должен быть ABS.

Предположим, например, что Вам требуется доступ к переменной по имени TOTAL, определенной в другом модуле. Тогда модуль, в котором определена переменная TOTAL, должен содержать операторы

```
        PUBLIC  TOTAL
TOTAL   DW      0       ;Присвоить TOTAL начальное значение 0
```

а модуль, который ссылается на переменную TOTAL, – оператор

```
EXTRN     TOTAL:WORD
```

Псевдооператор INCLUDE на время трансляции вставляет в текущий файл исходной программы целый *файл* исходных операторов. Например, оператор

```
INCLUDE    B:OTHERFIL.ASM
```

считывает содержимое файла OTHERFIL.ASM с диска B в Ваш исходный файл на свое место. Оператор INCLUDE можно использовать также для считывания в программу *макроопределений*. (Подробнее об этом см. в гл. 9.)

ПСЕВДООПЕРАТОРЫ УПРАВЛЕНИЯ ТРАНСЛЯЦИЕЙ

Существуют несколько псевдооператоров управления трансляцией (они описаны в разд. 2.8). Наиболее часто из них употребляется псевдооператор END.

Псевдооператор END отмечает конец исходной программы и указывает Ассемблеру, где завершить трансляцию. Поэтому псевдооператор END *должен присутствовать в каждой исходной программе*. Он имеет формат

```
END [метка точки входа]
```

где **метка точки входа** идентифицирует Вашу исходную программу. Например, оператор

```
END MY_PROG
```

отмечает конец программы MY_PROG.

Важно иметь в виду, что если Ваша программа состоит из нескольких модулей, то метка должна присутствовать только в операторе END основного модуля. В остальных модулях оператор END должен быть указан без метки. Например, если Ваша основная программа вызывает подпрограммы, находящиеся в отдельных модулях, то оператор END, завершающий каждый программный модуль, должен быть без метки.

ПСЕВДООПЕРАТОРЫ УПРАВЛЕНИЯ ЛИСТИНГОМ

Псевдооператор PAGE имеет формат

```
PAGE [число строк] [,число столбцов]
```

где **число строк** и **число столбцов** задают длину и ширину страниц листинга программы. Диапазон числа строк – от 10 до 255, числа столбцов – от 60 до 132. По умолчанию размер страницы составляет 57 строк по 80 символов. Например, оператор

```
PAGE 25,100
```

ограничивает размер каждой страницы 25 строками по 100 символов в каждой.

Обычно листинг выдается на стандартную бумагу с размером страницы 66 строк по 132 символа, и в этом случае оператор PAGE имеет вид

```
PAGE ,132
```

(Если Вы используете бумагу шириной 8,5 дюйма (216 мм), то установите на принтере режим печати самым узким шрифтом, обычно 16,5 символов на дюйм.)

На верхней строке каждой страницы Ассемблер печатает номер главы и номер страницы, разделяя их дефисом. Ассемблер переходит к следующей странице, если текущая страница заполнилась либо если ему встретился простой оператор PAGE (без операндов), и увеличивает номер главы только в том случае, если ему встретился оператор PAGE +, и при этом начинает нумерацию страниц заново с 1. В любом из этих трех случаев принтеру дается команда перехода на начало следующей страницы.

Псевдооператор TITLE обеспечивает печать заголовка на второй строке каждой страницы. Заголовок выравнивается слева; обычно в нем указывают имя файла с

Таблица 2.3. **Псевдооператоры управления листингом**

Псевдооператор	Функция
PAGE	*Формат*: PAGE **[число строк] [,число столбцов]** Устанавливает длину и ширину печатаемой страницы.
TITLE	*Формат*: TITLE **текст** Указывает заголовок, который должен быть напечатан на второй строке каждой страницы.
SUBTTL	*Формат*: SUBTTL **текст** Указывает подзаголовок, который должен быть напечатан на третьей строке каждой страницы.

исходным модулем и краткое описание назначения этого модуля. Чаще всего оператор TITLE помещают в самом начале программы, но он может быть указан в любом месте программы.

Псевдооператор SUBTTL обеспечивает печать центрированного подзаголовка на третьей строке каждой страницы; обычно в нем описывается содержание страницы. Например, начало листинга могло иметь такие заголовки:

```
TITLE COUNT_ALL - программа переписи Галактики
SUBTTL Сегмент данных Венеры
```

По завершении печати первого сегмента данных Вы могли бы изменить подзаголовок следующим образом:

```
SUBTTL Сегмент данных Плутона
PAGE
```

Заголовки и подзаголовки могут содержать до 60 символов.

В табл. 2.3 приведены все псевдооператоры управления листингом.

2.6. ОПЕРАЦИИ

Примечание. Этот раздел включен в основном для справок. Начинающим программистам рекомендуем пропустить его и вернуться к нему, если появится необходимость в дополнительных сведениях.

Операция является модификатором, который используется в операторе языка ассемблера или в псевдооператоре. Существует пять видов операций (табл.2.4): арифметические, логические, отношения, а также операции, возвращающие значения, и операции присваивания атрибута.

АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ

Арифметические операции производятся над числовыми операндами и дают числовой результат. Наиболее часто используются такие операции, как сложение (+), вычитание (–), умножение (*) и деление (/).

Операция деления (/) возвращает частное от деления нацело. Например, оператор

```
PI_QUOT EQU 31416/10000
```

возвратит значение 3.

У Ассемблера есть также операция MOD, возвращающая остаток от деления нацело. Оператор

Таблица 2.4. Операции

Операции	Функция
Арифметические	
+	*Формат*: **значение 1 + значение 2** Складывает **значение 1** и **значение 2**.
–	*Формат*: **значение 1 – значение 2** Вычитает **значение 2** из **значения 1**.
*	*Формат*: **значение 1 * значение 2** Умножает **значение 1** на **значение 2**.
/	*Формат*: **значение 1/значение 2** Делит нацело **значение 1** на **значение 2** и возвращает частное.
MOD	*Формат*: **значение 1** MOD **значение 2** Делит нацело **значение 1** на **значение 2** и возвращает остаток.
SHL	*Формат*: **значение** SHL **выражение** Сдвигает **значение** влево на число битов, равное значению **выражение**. Примечание. Не пользуйтесь операцией SHL при работе с Макроассемблером версии 1.00.
SHR	*Формат*: **значение** SHR **выражение** Сдвигает **значение** вправо на число битов, равное значению **выражение**. Примечание. Не пользуйтесь операцией SHR при работе с Макроассемблером версии 1.00.
Логические	
AND	*Формат*: **значение 1** AND **значение 2** Выполняет логическую операцию И над **значение 1** и **значение 2**.
OR	*Формат*: **значение 1** OR **значение 2** Выполняет логическую операцию ИЛИ над **значение 1** и **значение 2**.
XOR	*Формат*: **значение 1** XOR **значение 2** Выполняет логическую операцию ИСКЛЮЧАЮЩЕЕ ИЛИ над **значение 1** и **значение 2**.
NOT	*Формат*: NOT **значение** Обращает каждый бит в **значение**, иначе говоря, осуществляет дополнение до единицы.
Отношения	
EQ	*Формат*: **операнд 1** EQ **операнд 2** Истинно, если значения операндов совпадают.
NE	*Формат*: **операнд 1** NE **операнд 2** Истинно, если значения операндов не совпадают.
LT	*Формат*: **операнд 1** LT **операнд 2** Истинно, если **операнд 1** меньше **операнд 2**.
GT	*Формат*: **операнд 1** GT **операнд 2** Истинно, если **операнд 1** больше **операнд 2**.
LE	*Формат*: **операнд 1** LE **операнд 2** Истинно, если **операнд 1** меньше **операнд 2** или равен ему.
GE	*Формат*: **операнд 1** GE **операнд 2** Истинно, если **операнд 1** больше **операнд 2** или равен ему.

Операции	Функция
Возвращающие значения	
$	*Формат*: $ Возвращает значение счетчика текущей ячейки.
SEG	*Формат*: SEG **переменная** *или* SEG **метка** Возвращает номер блока адреса переменной или метки.
OFFSET	*Формат*: OFFSET **переменная** или OFFSET **метка** Возвращает смещение адреса переменной или метки.
LENGTH	*Формат*: LENGTH **переменная** Возвращает длину в единицах определения (байтах или словах) любой переменной, при определении которой был использован псевдооператор DUP.
TYPE	*Формат*: TYPE **переменная** *или* TYPE **метка** Для переменной операция TYPE возвращает 1, если она имеет тип BYTE, 2 (WORD), 4 (DOUBLEWORD). Для меток она возвращает –1 (атрибут NEAR) или –2 (атрибут FAR).
SIZE	*Формат*: SIZE **переменная** Возвращает произведение LENGTH и TYPE.
Присваивания атрибута	
PTR	*Формат*: **тип** PTR **выражение** Изменяет атрибут типа (BYTE или WORD) или атрибут дистанции (NEAR или FAR) адресного операнда. Здесь **тип** – новый атрибут, а **выражение** – идентификатор, чей атрибут должен быть изменен.
DS: ES: SS: CS:	*Формат*: **регистр_сег: адресное_выражение** *или* **регистр_сег: метка** *или* **регистр_сег: переменная** Изменяет атрибут сегмента метки, переменной или адресного выражения.
SHORT	*Формат*: JMP SHORT **метка** Изменяет атрибут NEAR метки оператора перехода JMP и указывает, что переход осуществляется на расстояние не более +127 байт или –128 байт от следующей команды.
THIS	*Формат*: THIS **атрибут** *или* THIS **тип** Создает адресный операнд либо с атрибутом дистанции (NEAR или FAR), либо с атрибутом типа (BYTE или WORD) со смещением, равным текущему значению счетчика адреса, и атрибутом текущего сегмента.

Операции	Функция
HIGH	*Формат*: HIGH значение *или* HIGH выражение Возвращает старший байт 16-битового числового значения или адресного выражения.
LOW	*Формат*: LOW значение *или* LOW выражение Возвращает младший байт 16-битового числового значения или адресного выражения.

```
PI_REM EQU 31416 MOD 10000
```

определит константу с именем PI_REM и значением 1416.

Наконец, операции SHL и SHR сдвигают числовой операнд влево и вправо. Как правило, они требуются для определения *масок*, которые будут использоваться в логических операциях. Например, если Вы определили маску оператором

```
MASK EQU 110010B
```

то оператор

```
MASK_LEFT_2 EQU MASK SHL 2
```

определит новую константу со значением 11001000B. Аналогично оператор

```
MASK_RIGHT_2 EQU MASK SHR 2
```

определит новую константу со значением 1100B.

ЛОГИЧЕСКИЕ ОПЕРАЦИИ

Подобно операциям SHL и SHR, описанным в предыдущем разделе, логические операции в основном используются для манипулирования двоичными, а не десятичными значениями. Однако в отличие от них логические операции манипулируют отдельными битами, а не группой битов.

Чтобы провести аналогию, представим себе группу пациентов, дожидающихся приема в поликлинике на длинной скамье. Медсестра может пригласить первых трех пациентов на осмотр и попросить остальных сдвинуться влево. Тем самым она фактически выполняет операцию SHL 3.

Но если прием в поликлинике организован *логически*, то медсестра может пригласить на осмотр определенных пациентов (скажем, имеющих переломы костей) и попросить остальных оставаться на своих местах.

Логические операции AND, OR и XOR выполняются над двумя операндами, а операция NOT – над одним (табл. 2.5).

Операция AND (И) удобна для фильтрации, маскирования или удаления некоторых битов. Она полагает бит результата равным 1 для каждой позиции, где оба операнда содержат 1. Для любой другой комбинации битов операндов операция AND обнуляет бит результата.

Например, операция

```
00110100B AND 11010111B
```

Таблица 2.5. **Действие операций AND, OR и XOR**

Операнд 1	Операнд 2	Результат		
		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

даст результат 00010100B. Как легко видеть, по своему действию операция AND похожа на дом с одной дверью и двумя замками. Если оба замка открыты (1 означает "открыт"), то в дом можно войти; если какой-то один или оба замка заперты (0 означает "заперт"), то Вы останетесь снаружи.

Операция OR (ИЛИ) полагает бит результата равным 1 для каждой позиции, где хотя бы один из двух операндов содержит 1. В позициях, где оба операнда содержат 0, операция OR обнуляет бит результата.

Для предыдущих операндов операция

```
00110100B OR 11010111B
```

даст результат 11110111B. Продолжая аналогию с дверью, можно сказать, что по своему действию операция OR похожа на дом с двумя дверями. Если хотя бы одна из них открыта (1), то в дом можно войти, но если обе заперты (0), то Вы останетесь снаружи.

Операция XOR (исключающее ИЛИ) представляет собой модификацию операции OR, которая вместо установки бита результата в 1 обнуляет его для каждой позиции, где оба операнда содержат 1. Такое название этой операции дано потому, что она исключает комбинацию битов 1-1 (в отличие от названия "включающее ИЛИ").

Операция NOT инвертирует каждый бит операнда. Другими словами, она заменяет 1 на 0 и 0 на 1.

Например, операция

```
NOT 01101001B
```

дает результат 10010110B.

Среди команд языка ассемблера для микропроцессора 8088 есть одноименные операциям AND, OR, XOR и NOT. Они обсуждаются в гл.3. Разница в том, что эти логические команды действуют на этапе исполнения программы, а логические операции – на этапе ее *трансляции*.

ОПЕРАЦИИ ОТНОШЕНИЯ

Операции отношения сравнивают два числовых значения или два адреса памяти из одного сегмента и дают числовой результат. Результатом может быть одно из двух чисел: 0, если отношение "ложно", и 0FFFFH, если оно "истинно".

Например, оператор

```
MOV AX,CHOICE LT 20
```

где CHOICE – ранее определенная константа, будет во время трансляции заменен на оператор

```
MOV AX,OFFFFH
```

если значение CHOICE меньше 20, или на оператор

```
MOV AX,0
```

если значение CHOICE больше или равно 20.

Так как операции отношения дают только два значения (0 и 0FFFFH), то они редко используются сами по себе. Чаще всего они применяются в сочетании с другими операциями для формирования выражений "принятия решения". Например, пусть Вам требуется загрузить в регистр AX значение 5, если значение CHOICE меньше 20, и 6 в противном случае. Такую задачу выполнит оператор

```
MOV AX,((CHOICE LT 20) AND 5) OR ((CHOICE GE 20) AND 6
```

Действительно, если значение CHOICE меньше 20, то выражение (CHOICE LT 20) "истинно", а выражение (CHOICE GE 20) "ложно". Поэтому промежуточная форма приведенного выше оператора будет иметь вид

```
MOV AX,(OFFFFH AND 5) OR (O AND 6)
```

Ассемблер воспримет этот оператор как оператор

```
MOV AX,5
```

С другой стороны, если значение CHOICE больше или равно 20, то выражение (CHOICE LT 20) "ложно", выражение (CHOICE GE 20) "истинно". Поэтому промежуточная форма примет вид

```
MOV AX,(O AND 5) OR (OFFFFH AND 6)
```

и Ассемблер воспримет ее как оператор

```
MOV AX,6
```

ОПЕРАЦИИ, ВОЗВРАЩАЮЩИЕ ЗНАЧЕНИЯ

Операции этой группы предоставляют информацию о переменных или метках программы.

Операция,обозначенная знаком доллара ($),возвращает текущее значение счетчика адреса, иначе говоря, смещение адреса текущего оператора. Эта операция полезна, если Вы хотите заставить Ассемблер вычислить длины строк символов. Например, при трансляции операторов

```
MESSAGE DB  Для продолжения работы нажмите любую клавишу
MESSAGEL EQU $-MESSAGE
```

Ассемблер определит число символов в строке MESSAGE и присвоит его константе MESSAGEL. При выдаче сообщения на экран мы можем использовать значение константы MESSAGEL для задания числа выдаваемых символов.

Операции SEG и OFFSET возвращают значения номера блока и смещения адреса переменной или метки. Например, операторы

```
MOV AX,SEG    TABLE
MOV BX,OFFSET TABLE
```

загрузят номер блока и смещение адреса переменной TABLE в регистры AX и BX соответственно. Конечно, так как и номер блока, и смещение имеют 16-битовые значения, то они должны загружаться в 16-битовые регистры.

Операция TYPE возвращает числовое значение, идентифицирующее тип атрибута переменной или тип атрибута дистанции метки. Для переменной операция TYPE возвращает 1, если переменная имеет тип BYTE, и 2, если она имеет тип WORD. Для метки операция TYPE возвращает -1, если она имеет атрибут NEAR, и -2, если она имеет атрибут FAR.

Операции LENGTH (длина) и SIZE (размер) полезны только для переменных, при определении которых была использована операция дублирования DUP. Операция LENGTH возвращает число единиц определения (байтов или слов) в зарезервированной для переменной памяти. Например, операторы

```
TABLE DW 100 DUP(1)
      MOV CX,LENGTH TABLE ;Загрузить в CX число слов в таблице TABLE
```

загрузят 100 в регистр CX. Если Вы используете операцию LENGTH для переменной, в определении которой операция DUP не участвовала, то операция LENGTH возвратит 1.

Операция SIZE возвращает число байтов, зарезервированных для переменной, иначе говоря, произведение операций LENGTH и TYPE. Для определенной выше переменной TABLE оператор

```
MOV CX,SIZE TABLE ;Загрузить в CX число байтов в таблице TABLE
```

загрузит 200 в регистр CX.

ОПЕРАЦИИ ПРИСВАИВАНИЯ АТРИБУТОВ

Операция указателя PTR позволяет изменить у операнда атрибут типа (BYTE или WORD) или атрибут дистанции (NEAR или FAR). Например этой операцией можно воспользоваться для доступа к байтам в таблице слов. Если таблица определена следующим образом:

```
WORD_TABLE DW 100 DUP(?)
```

то оператор

```
FIRST_BYTE EQU BYTE PTR WORD_TABLE
```

присвоит имя ячейке первого байта таблицы WORD_TABLE. Можно присвоить имя и любому другому байту, например

```
FIFTH_BYTE EQU FIRST_BYTE+4
```

Как уже упоминалось, операция PTR может изменить и атрибут дистанции метки. Например, если программа содержит оператор

```
START: MOV CX,100
```

то метка START имеет атрибут NEAR, что позволяет ссылаться на нее команде перехода JMP, находящейся в том же сегменте. Чтобы на эту метку могли ссылаться команды JMP, находящиеся в других сегментах, надо дать приведенному выше оператору альтернативную метку, имеющую атрибут FAR. Это можно сделать оператором

```
FAR_START EQU FAR PTR START
```

Как упоминалось в гл. 1, при вычислении адреса микропроцессор 8088 автоматически выбирает регистр сегмента SS, если смещение операнда находится в регистрах SP или BP. Аналогично он выбирает регистр сегмента DS, если смещение содержится в регистрах BX, SI или DI. Операция *замены сегмента* (DS:, ES:, SS: или CS:) позволяет изменить атрибут сегмента метки, сегмента или адресного выражения. Например, оператор

```
MOV AX,ES:[BP]
```

сообщает микропроцессору 8088, что операнд-источник надо извлечь из дополнительного сегмента, а не из сегмента стека.

Операция SHORT сообщает Ассемблеру, что адрес перехода команды JMP находится не далее +127 или −128 байт от следующей команды. Имея такую информа-

цию, Ассемблер закодирует команду JMP двумя байтами, а не тремя, что сэкономит память. Приведем пример:

```
        JMP SHORT THERE
        ...
        ...
THERE:
```

Операция THIS создает адресный операнд с заданным атрибутом типа BYTE или WORD либо дистанции NEAR или FAR и определяет для него те же атрибуты сегмента и смещения, которые должны быть у адреса следующей доступной ячейки памяти. Например, последовательность операторов

```
FIRST_BYTE EQU THIS BYTE
WORD_TABLE DW 100 DUP(?)
```

создает адресную константу FIRST_BYTE со значением, равным адресу переменной WORD_TABLE, и приписывает ей атрибут BYTE. Она выполняет ту же функцию, что и ранее рассмотренный оператор

```
FIRST_BYTE EQU BYTE PTR WORD_TABLE
```

С помощью операции THIS можно приписать ячейке с командой атрибут FAR. Модифицируем один из предыдущих примеров: при трансляции операторов

```
START EQU THIS FAR
      MOV CX,100
```

команда MOV получит атрибут FAR, что позволит находящимся в других сегментах командам JMP обеспечить прямой переход к метке START.

Операции HIGH и LOW возвращают соответственно старший и младший байты 16-битового выражения. Например, если константа определена оператором

```
CONST EQU OABCDH
```

то оператор

```
MOV AH,HIGH CONST
```

загрузит в регистр AH значение 0ABH.

2.7. ВВОД, ТРАНСЛЯЦИЯ И ИСПОЛНЕНИЕ ПРОГРАММЫ

Так как мы еще не обсуждали детали системы команд языка ассемблера для микропроцессора 8088 (они будут описаны в гл.3), то пока Вы не сможете написать программу, выполняющую операции сложения и вычитания или манипулирующую регистрами, или выполняющую множество других видов деятельности, которые Вы хотите "поручить" ЭВМ. Тем не менее *полученных* Вами сведений вполне достаточно, чтобы написать программу для пересылки данных с помощью команд MOV и определения необходимых сегментов с помощью псевдооператоров.

Мы посвятим этот раздел разработке программы, которая копирует данные из одной четырехбайтовой таблицы в другую. Чтобы сделать задачу более интересной, будем заполнять данные в "таблице-приемнике" в обратном порядке по отношению к их расположению в "таблице-источнике".

В любом случае детали программы не имеют значения. На этом примере Вы должны в основном научиться вводу программы в ЭВМ, выполнению ее трансляции, получению файла с листингом и исполняемого модуля, а также исполнению и отладке программы. Короче говоря, Вы должны сами пройти основные этапы. Это

поможет приобрести уверенность в себе для освоения более сложного материала, содержащегося в следующих главах.

СОЗДАНИЕ РАБОЧЕГО ДИСКА АССЕМБЛЕРА

При программировании на языке ассемблера потребуется ряд программ, часть которых находится на дисках с операционной системой DOS фирмы IBM (DOS версии 1.1 записана на одном диске), а остальные на диске с Макроассемблером. Если у Вашей ЭВМ есть жесткий диск, просто скопируйте содержимое диска с Макроассемблером в ту область файлов, которая содержит DOS. Если же Ваша ЭВМ снабжена только гибкими дисками, то Вам придется подготовить диск, содержащий все необходимые программы. (Это имеет смысл только в том случае, если дисководы Вашей ЭВМ позволяют работать с двусторонними дисками, так как на одностороннем диске все необходимые программные файлы не поместятся.)

Для начала с помощью команды FORMAT/S операционной системы DOS выполните разметку чистого диска. Затем скопируйте не него файлы EDLIN.COM с основного диска DOS и DEBUG.COM с диска дополнительных программ DOS. В зависимости от того, какой версией Макроассемблера Вы располагаете, сделайте следующие действия:

- если у Вас версия 1, то скопируйте файл LINK.EXE с диска дополнительных программ DOS, а затем скопируйте файлы MASM.EXE и CREF.EXE с диска с Макроассемблером;

если у Вас версия 2, то скопируйте файлы MASM.EXE, CREF.EXE, SALUT.EXE и LIB.EXE с диска с Макроассемблером. Затем скопируйте файл LINK.EXE с диска с дополнительными программами DOS (для операционной системы DOS 1.1) или с диска с Макроассемблером (для операционной системы DOS версии 2.0 и более поздних версий).

Созданный таким образом диск будем называть *диском Ассемблера*.

ДИСК ДАННЫХ

Для выполнения описанных в этом разделе процедур Вам понадобится еще один чистый диск, который надо разместить с помощью простой команды FORMAT (без /S). Он будет служить для Вас диском данных.

ПРИМЕР ПРОГРАММЫ

На рис. 2.1 показан текст нашей программы копирования таблицы. Обратите внимание, что она имеет сегмент стека STACK, сегмент данных DSEG и сегмент команд CSEG. Сегмент стека будет содержать адрес возврата, который позволит микропроцессору 8088 вернуться к отладчику DEBUG после завершения программы.

Сегмент данных состоит только из двух псевдооператоров. Один определяет таблицу-источник SOURCE, а другой резервирует место в памяти для таблицы-приемника DEST.

Сегмент команд содержит четыре группы команд:

первая группа помещает адрес возврата отладчика DEBUG в стек;

вторая группа заставляет регистр DS указывать на сегмент данных (*напомним, что оператор ASSUME об этом не заботится*);

третья группа обнуляет четыре байта таблицы DEST. Это делается для того,

чтобы при анализе конечного состояния таблицы DEST исключить возможность влияния предыдущих исполнений программы;

четвертая группа копирует данные таблицы по одному байту за прием.

Команды, находящиеся в сегменте команд, образуют единственную процедуру, окаймленную псевдооператорами OUR_PROG PROC FAR и OUR_PROG ENDP. Этой процедуре присвоен атрибут FAR, поэтому при завершении исполнения программы микропроцессор 8088 по команде RET возвратит управление отладчику DEBUG (находящемуся в другом сегменте памяти). *Из этих соображений Вам следует оформлять свои программы как процедуры.*

```
TITLE   EX_PROG   -   Пример программы
          PAGE        ,132
STACK     SEGMENT    PARA STACK 'STACK'
         DB     64 DUP('STACK    ')
STACK       ENDS
DSEG        SEGMENT  PARA PUBLIC 'DATA'
SOURCE      DB    10,20,30,40     ;Эта таблица будет скопирована в
DEST        DB    4 DUP(?)        ;эту таблицу, в обратном порядке
DSEG        ENDS
SUBTTL  Основная программа
            PAGE
CSEG        SEGMENT PARA PUBLIC 'CODE'
OUR_PROG    PROC     FAR
            ASSUME   CS:CSEG,DS:DSEG,SS:STACK
;
;  Занести в стек такие начальные значения, чтобы программа
;  могла возвратить управление отладчику DEBUG
;
            PUSH   DS              ;Поместить в стек номер блока адреса возврата
            MOV    AX,0            ;Обнулить регистр
            PUSH   AX              ;Поместить в стек нулевое смещение адреса
;                                    возврата
;
;  Инициировать адрес сегмента данных
;
            MOV    AX,DSEG         ;Инициировать DS
            MOV    DS,AX
;
;  Присвоить элементам таблицы DEST нулевые начальные значения
;
            MOV    DEST,0          ;Первый байт
            MOV    DEST+1,0        ;Второй байт
            MOV    DEST+2,0        ;Третий байт
            MOV    DEST+3,0        ;Четвертый байт
;
;  Скопировать таблицу SOURCE в таблицу DEST, в обратном порядке
;
            MOV    AL,SOURCE       ;Скопировать первый байт
            MOV    DEST+3,AL
            MOV    AL,SOURCE+1     ;Скопировать второй байт
            MOV    DEST+2,AL
            MOV    AL,SOURCE+2     ;Скопировать третий байт
            MOV    DEST+1,AL
            MOV    AL,SOURCE+3     ;Скопировать четвертый байт
            MOV    DEST,AL
            RET                    ;Возвратить управление отладчику DEBUG
OUR_PROG    ENDP
CSEG        ENDS
            END    OUR_PROG
```

Рис. 2.1. Пример программы для ввода и трансляции

Для ввода программы в ЭВМ надо воспользоваться либо программой EDLIN, либо (что еще лучше) любым другим редактором или программой обработки текста, которые сохраняют текст в виде стандартных кодов ASCII (т. е. в неформатированном виде без управляющих символов). Например, Вы можете воспользоваться программой WordStar в режиме "не-документа" N. Так как нам не известно, какая программа обработки текста есть у Вас (и есть ли вообще), то будем предполагать, что Вы пользуетесь программой EDLIN. В табл. 2.6 описаны наиболее полезные команды этой программы.

Вообще говоря, мы будем исходить из того, что у Вас есть два дисковода для гибких дисков. Если у Вас только один дисковод, то описанной ниже процедурой можно пользоваться за счет частых перестановок дисков. Если Вы располагаете жестким диском, то игнорируйте приглашение к вводу A> и B>, которые относятся к гибким дискам. Вместо этого подразумевайте C> всюду, где в книге указано A> или B>.

Для ввода программы действуйте следующим образом:

1. Вставьте Ваш новый диск с Ассемблером в левый дисковод (A), а чистый диск данных – в правый дисковод (B), затем включите питание ЭВМ. Нажмите клавишу возврата каретки, когда ЭВМ запросит ввод даты и времени.

2. Когда появится приглашение к вводу A>, наберите b: и нажмите клавишу возврата каретки. Тем самым активным станет правый дисковод. (На будущее запомните, что для завершения ввода любой команды надо нажимать клавишу возврата каретки.)

3. Введите команду

```
B>a:edlin ex_prog.asm
```

Здесь ex_prog – имя программы, которую мы хотим создать. Расширение имени (.asm) указывает, что это исходная программа на языке ассемблера.

4. Когда ЭВМ выдаст на экран

```
New file
*_
```

(новый файл), нажмите I для перевода программы EDLIN в режим вставки. Следующее приглашение к вводу

```
1:*
```

означает, что программа EDLIN ждет ввода первой строки текста – в нашем случае, первого оператора исходной программы.

5. Поочередно введите строки программы, изображенной на рис. 2.1. Мы набрали большинство слов прописными буквами, но если хотите, можете набирать вместо них строчные буквы. Кроме того, для большего удобства чтения листинга мы выравнивали поля операторов, но Вы можете вводить их как заблагорассудится. Надо лишь не забывать вставлять между полями хотя бы один пробел.

6. По окончании ввода наберите Ctrl-Break[1]. Это заставит программу EDLIN выйти из режима вставки. Затем наберите E для сохранения программы на диске данных.

Теперь Вы можете оттранслировать *исходную программу* и получить *объектную программу*.

---

[1] То есть нажмите клавишу управляющего регистра Ctrl и, удерживая ее, нажмите клавишу Break. – *Прим. перев.*

Таблица 2.6. **Общеупотребительные команды программы EDLIN**

Команда	Действие	Примечание
[нач_строка] [, кон_строка] D	Удалить строку	Ввод команды D без параметров приводит к уничтожению текущей строки
[строка]	Изобразить редактируемую строку	Нажатие клавиши возврата каретки вызывает переход к следующей строке
E	Сохранить программу на диске и вернуться к DOS	Используйте E для завершения сеанса редактирования (см. ниже команду Q)
[строка] I	Вставить набираемые на клавиатуре строки перед указанной строкой. Для выхода из режима вставки нажмите Ctrl-Break	При наборе новой исходной программы укажите I без номера строки
[нач_строка] [, кон_строка] L	Напечатать (изобразить) строки	
Q	Вернуться к DOS без сохранения отредактированного текста	См. выше команду E
[нач_строка], [кон_строка], адресат C	Вставить копии указанных строк непосредственно перед строкой **адресат**	Имеется в DOS 2.0 и более поздних версиях
[нач_строка], [кон_строка], адресат M	Переместить указанные строки, вставляя их непосредственно перед строкой **адресат**	Имеется в DOS 2.0 и более поздних версиях
[нач_строка], [кон_строка] R **[старый_фрагмент]** затем <F6> **[новый_фрагмент]**	Заменить образец **старый_фрагмент** на **новый_фрагмент** в указанном диапазоне строк	
[нач_строка], [кон_строка] S **фрагмент**	Найти образец **фрагмент** в указанном диапазоне строк	

Примечание. В квадратные скобки заключены необязательные элементы команд.

ТРАНСЛЯЦИЯ ПРОГРАММЫ

Для трансляции программы наберите

```
B>a:masm ex_prog
```

Ассемблер выдаст три запроса, на которые надо ответить так:

```
Object filename [EX_PROG.OBJ]: (нажмите клавишу возврата каретки)
Source listing [NUL.LST]: ex_prog
Cross reference [NUL.CRF]: (нажмите клавишу возврата каретки)
```

Таким образом, Ассемблеру дается указание обработать исходный файл EX_PROG. ASM для создания объектного файла с предложенным Вам именем EX_PROG. OBJ и выдать листинг трансляции в файл EX_PROG. LST. Листинг трансляции содержит команды исходной программы и соответствующие им числовые коды. Эта полезная информация показывает, как Ассемблер интерпретирует Вашу программу.

Если Вы аккуратно выполнили предшествующие шаги, то Ассемблер завершит свою работу сообщением

```
Warning          Severe
Errors           Errors
0                0
```

что означает

```
(Предупреж-      (Серьезных
дений            ошибок
0)               0)
```

и возвратит управление операционной системе DOS. Если же Ассемблер выдал сообщения об ошибках, то исправьте исходную программу с помощью редактора EDLIN и заново ее оттранслируйте.

ЛИСТИНГ ИСХОДНОЙ ПРОГРАММЫ

Для выдачи листинга трансляции на экран введите команду

```
B>type ex_prog.lst
```

Команды программы промелькнут на экране так быстро, что Вы не успеете их рассмотреть. Для приостанова выдачи можно набрать Ctrl-NumLock; для продолжения – произвольную клавишу. Для получения распечатки изображения листинга на экране переведите свой принтер в режим on-line и наберите Ctrl-PrtSc.

Листинг должен выглядеть так, как показано на рис. 2.2. Обратите внимание на то, что благодаря псевдооператору PAGE, указанному в исходной программе, листинг занимает не две страницы, а три.

На первых двух страницах распечатаны команды исходной программы и соответствующие им объектные коды в следующем формате:

левый столбец листинга содержит шестнадцатеричные значения смещения адреса (в байтах) от начала сегмента;

последующие столбцы чисел содержат объектный код каждого оператора. Для сегмента стека и сегмента данных эти числа показывают значения, запоминаемые в каждой ячейке памяти. Для сегмента команд эти числа означают машинные коды, выполняемые микропроцессором 8088;

текст в правой части листинга взят из исходной программы.

```
 IBM Personal Computer MACRO Assembler   Version 2.00     Page    1-1
EX_PROG  -  Пример программы                                  12-17-84


                                  TITLE  EX_PROG  -  Пример программы
                                         PAGE      ,132
0000                              STACK     SEGMENT   PARA STACK 'STACK'
0000    40[                                 DB    64 DUP('STACK   ')
          53 54 41 43 4B
             20 20 20
                          ]

0200                              STACK      ENDS
0000                              DSEG       SEGMENT  PARA PUBLIC 'DATA'
0000  0A 14 1E 28                 SOURCE     DB   10,20,30,40    ;Эта таблица будет скопирована в
0004    04[                       DEST       DB   4 DUP(?)       ;эту таблицу, в обратном порядке
          ??
                          ]

000B                              DSEG       ENDS
                                  SUBTTL  Основная программа
```

Рис. 2.2. Листинг трансляции примера программы

```
                            Основная программа

                                   PAGE
0000                        CSEG   SEGMENT PARA PUBLIC 'CODE'
0000                        OUR_PROG  PROC   FAR
                                   ASSUME  CS:CSEG,DS:DSEG,SS:STACK
                            ;
                            ;  Занести в стек такие начальные значения, чтобы программа
                            ;  могла возвратить управление отладчику DEBUG
                            ;
0000  1E                           PUSH  DS          ;Поместить в стек номер блока адреса возврата
0001  B8 0000                      MOV   AX,0        ;Обнулить регистр
0004  50                           PUSH  AX          ;Поместить в стек нулевое смещение адреса
                            ;                          возврата
                            ;
                            ;  Инициировать адрес сегмента данных
                            ;
0005  B8 ---- R                    MOV   AX,DSEG     ;Инициировать DS
0008  8E D8                        MOV   DS,AX
                            ;
                            ;  Присвоить элементам таблицы DEST нулевые начальные значения
                            ;
000A  C6 06 0004 R 00              MOV   DEST,0      ;Первый байт
000F  C6 06 0005 R 00              MOV   DEST+1,0    ;Второй байт
0014  C6 06 0006 R 00              MOV   DEST+2,0    ;Третий байт
0019  C6 06 0007 R 00              MOV   DEST+3,0    ;Четвертый байт
                            ;
                            ;  Скопировать таблицу SOURCE в таблицу DEST, в обратном порядке
                            ;
001E  A0 0000 R                    MOV   AL,SOURCE   ;Скопировать первый байт
0021  A2 0007 R                    MOV   DEST+3,AL
0024  A0 0001 R                    MOV   AL,SOURCE+1 ;Скопировать второй байт
0027  A2 0006 R                    MOV   DEST+2,AL
002A  A0 0002 R                    MOV   AL,SOURCE+2 ;Скопировать третий байт
002D  A2 0005 R                    MOV   DEST+1,AL
0030  A0 0003 R                    MOV   AL,SOURCE+3 ;Скопировать четвертый байт
0033  A2 0004 R                    MOV   DEST,AL
0036  CB                           RET               ;Возвратить управление отладчику DEBUG
0037                        OUR_PROG  ENDP
0037                        CSEG   ENDS
                                   END   OUR_PROG
```



```
Segments and Groups:

              N a m e          Size   Align   Combine Class

CSEG . . . . . . . . . . . . .  0037   PARA    PUBLIC  'CODE'
DSEG . . . . . . . . . . . . .  0008   PARA    PUBLIC  'DATA'
STACK  . . . . . . . . . . . .  0200   PARA    STACK   'STACK'

Symbols:

              N a m e          Type    Value   Attr

DEST . . . . . . . . . . . . .  L BYTE  0004    DSEG    Length = 0004
OUR_PROG . . . . . . . . . . .  F PROC  0000    CSEG    Length = 0037
SOURCE . . . . . . . . . . . .  L BYTE  0000    DSEG

50092 Bytes free

Warning Severe
Errors  Errors
0       0

A>
```

На третьей странице листинга дается детальная сводная информация о сегментах и идентификаторах программы. В принципе, эту часть листинга можно игнорировать. В Ассемблере версии 2 предусмотрен режим /N, отменяющий выдачу этих таблиц.

ПОИСК ТОЧКИ ОСТАНОВА ПРОГРАММЫ

Самое главное, что нам требуется от этого листинга, – это смещение адреса команды RET, указанной в конце программы. Мы используем его значение позже для исполнения программы вплоть до команды RET. После этого мы проверим содержимое регистров и таблиц данных. Нам надо остановиться непосредственно перед командой RET, поскольку после ее выполнения микропроцессор 8088 возвращает управление обратно в системную программу, откуда не так-то легко посмотреть эти значения.

Изобразите листинг на экране с помощью команды TYPE. Когда появится команда RET, то обратите внимание, что смещение ее адреса (крайнее левое число) равно 0036. Запомните его на будущее.

СОЗДАНИЕ ИСПОЛНЯЕМОГО ФАЙЛА

Операционная система DOS может считывать программу в любое подходящее место памяти. Чтобы воспользоваться этим, надо создать *перемещаемый* исполняемый файл.

Программа, создающая перемещаемый исполняемый файл, называется *загрузчиком*, поскольку может загрузить несколько объектных файлов в один большой исполняемый файл. Для вызова загрузчика введите команду

```
B>a:link ex_prog;
```

Когда вновь появится B>, на Вашем диске данных уже будет находиться исполняемый файл EX_PROG.EXE.

ЗАГРУЗКА НЕСКОЛЬКИХ ОБЪЕКТНЫХ МОДУЛЕЙ

В данном примере был только один объектный модуль EX_PROG.OBJ; при чтении других глав Вам придется создавать программы из двух или большего числа объектных модулей. В этом случае модули должны быть оттранслированы отдельно, а при запуске загрузчика имена объектных модулей надо указать, соединив их знаком + . Например, по команде

```
B>a:link mod1+mod2+mod3
```

загрузчик создаст исполняемый файл с именем MOD1.EXE из объектных модулей MOD1.OBJ, MOD2.OBJ и MOD3.OBJ.

ИСПОЛНЕНИЕ ПРОГРАММЫ

Как уже упоминалось, Вы можете вызвать свой перемещаемый исполняемый файл EX_PROG.EXE из операционной системы DOS или из отладчика DEBUG. Обычно программа вызывается из DOS только тогда, когда есть уверенность в ее правильности или когда она выдает какие-либо видимые результаты. В нашем примере программы нет ошибок (постучим по дереву!), но ее результаты остаются в памяти ЭВМ и не видны. Поэтому мы должны исполнять ее под управлением отладчика DEBUG. В табл. 2.7 приведены наиболее распространенные команды этого отладчика.

Таблица 2.7 **Общеупотребительные команды отладчика DEBUG**

Команда	Действие	Примечание
D адрес или **D адрес_начала смещение_конца**	Изобразить содержимое ячеек памяти	Обратите внимание, что для конечного адреса задается только смещение
E адрес значение 1 [значение 2. . .]	Изменить содержимое ячеек памяти	
F адрес_начала L значение_байта	Заполнить блок памяти заданным значением	
G **[смещение 1] [смещение 2. . .]**	Исполнить программу Значения смещений задают контрольные точки. Если они указаны, то процессор остановится перед выполнением команды в очередной контрольной точке и изобразит содержимое регистров.	См. ниже команду T
Q	Выйти из отладчика и вернуться к DOS.	
R **[имя-регистра]**	Изобразить содержимое одного или всех регистров	Если изображено содержимое одного регистра, то R позволяет Вам изменить его
T [число-команд]	Исполнить заданное число команд и изобразить содержимое регистров на каждом шаге	См. выше команду G
U [адрес]	Ретранслировать содержимое ячейки памяти в команду на языке ассемблера	

Примечание. В квадратные скобки заключены необязательные элементы команд.

Для начала введем команду

```
a:debug ex_prog.exe
```

После этого отладчик DEBUG должен выдать свое приглашение к вводу – знак дефиса (-).

На рис. 2.3 изображен типичный сеанс работы с отладчиком DEBUG. Проследим за каждым шагом работы. Не обращайте внимания на конкретные адреса: на Вашей машине они могут быть другими.

Вначале наберем R для изображения содержимого регистров. Команда отладчика R (register – регистр) показывает содержимое каждого регистра в виде четырех шестнадцатеричных цифр. Она также расшифровывает значения битов регистра флагов в виде ряда двухбуквенных мнемокодов. Их значения приведены в табл. 2.8. В нашем случае выданные командой R мнемокоды показывают, что в начальном состоянии все биты регистра флагов сброшены (т. е. имеют значения 0).

```
A>debug ex-prog.exe
-r

AX=0000  BX=0000  CX=0247  DX=0000  SP=0200  BP=0000  SI=0000  DI=0000
DS=10CB  ES=10CB  SS=10D8  CS=10F9  IP=0000   NV UP EI PL NZ NA PO NC
10F9:0000 1E            PUSH    DS
-t


AX=0000  BX=0000  CX=0247  DX=0000  SP=01FE  BP=0000  SI=0000  DI=0000
DS=10CB  ES=10CB  SS=10D8  CS=10F9  IP=0001   NV UP EI PL NZ NA PO NC
10F9:0001 B80000        MOV     AX,0000
-t


AX=0000  BX=0000  CX=0247  DX=0000  SP=01FE  BP=0000  SI=0000  DI=0000
DS=10CB  ES=10CB  SS=10D8  CS=10F9  IP=0004   NV UP EI PL NZ NA PO NC
10F9:0004 50            PUSH    AX
-g36


AX=102B  BX=0000  CX=0247  DX=0000  SP=01FC  BP=0000  SI=0000  DI=0000
DS=10FB  ES=10CB  SS=10D8  CS=10F9  IP=0036   NV UP EI PL NZ NA PO NC
10F9:0036 CB            RETF
-dds:0

10F8:0000  0A 14 1E 28 28 1E 14 0A-00 00 00 00 00 00 00 00   ...((.........
10FB:0010  1E BB 00 00 50 B8 FB 10-BE DB C6 06 04 00 00 C6   ....P.........
10FB:0020  06 05 00 00 C6 06 06 00-00 C6 06 07 00 00 A0 00   ..............
10FB:0030  00 A2 07 00 A0 01 00 A2-06 00 A0 02 00 A2 05 00   ..............
10FB:0040  A0 03 00 A2 04 00 CB 32-B5 50 EB 61 54 B3 C4 02   .......2.P.aT.
10FB:0050  C6 06 49 07 00 BB 46 FA-BB 56 FC A3 A0 07 B9 16   ..I...F..V....
10FB:0060  A2 07 E9 00 FF 90 B0 BE-F2 FB 00 74 0A B0 3E DE   ...........t..
10FB:0070  25 00 74 03 E9 B5 FE EB-5A FA 5E 5F BB E5 5D C3   %.t.....Z...].
-dss:0
10D8:0000  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
10D8:0010  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
10D8:0020  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
10D8:0030  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
10D8:0040  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
10D8:0050  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
10D8:0060  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
10D8:0070  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
-q

A>
```

Рис. 2.3. Сеанс исполнения примера программы с помощью отладчика DEBUG

Таблица 2.8. **Мнемокоды значений флагов, используемые отладчиком DEBUG**

Имя флага	Установлен	Сброшен
Переполнение (да/нет)	OV	NV
Направление (уменьшение/увеличение)	DN	UP
Прерывания (включены/отключены)	EI	DI
Знак (отрицательный/положительный)	NG	PL
Нуль (да/нет)	ZR	NZ
Вспомогательный перенос (да/нет)	AC	NA
Четность (чет/нечет)	PE	PO
Перенос (да/нет)	CY	NC

Последняя строка, выданная командой R, показывает, какую команду микропроцессор 8088 выполнит следующей (а не только что выполненную команду). Читая слева направо, мы видим, что следующая команда имеет адрес 0923: 0000 (это соответственно значения регистра сегмента команд CS и указателя команд IP), объектный код этой команды – 1E, а самая команда – PUSH DS. Пока что всё в порядке, поскольку PUSH DS и в самом деле первая команда в нашем примере программы.

Затем мы воспользуемся командами T (Trace – трассировка или пошаговое исполнение) для исполнения команды PUSH DS и следующей команды MOV AX, 0000. При каждом вводе команды T ЭВМ исполняет одну команду программы, затем изображает на экране содержимое регистров и показывает, какая команда будет исполнена следующей. Таким образом, трассировка позволяет Вам посмотреть, что содержат регистры в каждый момент времени и по какому пути идет исполнение программы. При отладке эта информация просто неоценима.

Обратите внимание на изменения содержимого регистров при исполнении каждой команды. После первой команды T в результате исполнения команды PUSH указатель стека SP изменит значение с 0200 на 01FE. Если бы при трассировке была исполнена команда PUSH AX, то Вы вновь обнаружили бы уменьшение значения регистра SP.

После трассировки двух первых команд мы воспользовались командой G (Go – прогнать до контрольной точки), чтобы продолжить исполнение программы вплоть до команды RET. При этом мы ввели g 36, поскольку, как мы заметили ранее, команда RET имеет смещение 36 от начала сегмента команд. Отладчик DEBUG изображает команду RET как RETF, поскольку ему придется делать возврат из процедуры с атрибутом FAR.

После завершения исполнения программы нам надо проверить, правильно ли она работала; другими словами, скопировала ли она в обратном порядке содержимое таблицы SOURCE в таблицу DEST. Чтобы увидеть конечное состояние таблиц, воспользуемся командой D (Dump – дамп). Введем

```
d ds:0
```

Эта команда сообщает отладчику DEBUG, что надо изобразить (d) содержимое сегмента данных (DS) с самого его начала (смещение 0).

В верхней строке экрана содержатся значения

```
0A 14 1E 28 28 1E 14 0A
```

Первые четыре значения представляют собой содержимое таблицы SOURCE; следующие четыре – содержимое таблицы DEST. Эти шестнадцатеричные числа представляют десятичные значения

```
10 20 30 40 40 30 20 10
```

Как видите, программа работает правильно.

Кроме изображения числовых значений ячеек памяти, команда D показывает также их символьные эквиваленты. В нашем случае это позволяет увидеть неиспользованную память в нижней части стека, которая обозначена повторением слова STACK в правой части дампа. Мы добились этого за счет того, что определили сегмент стека с помощью псевдооператора

```
DB 64 DUMP ('STACK   ')
```

И, наконец, по команде Q (Quit – завершить) мы выходим из отладчика DEBUG и возвращаемся в операционную систему DOS.

ДРУГИЕ ВИДЫ ЛИСТИНГА

При работе над длинной или сложной программой можно получить два других вида листинга с дополнительной информацией: *листинг перекрестных ссылок и листинг распределения памяти*.

ЛИСТИНГ ПЕРЕКРЕСТНЫХ ССЫЛОК

Листинг перекрестных ссылок указывает номер строки, в которой определен каждый идентификатор, и номера тех строк, в которых есть ссылки на него. Перед генерацией листинга перекрестных ссылок надо получить файл перекрестных ссылок с расширением имени (.CRF). Для этого надо в ответ на запрос Ассемблера "Cross reference" ввести имя программы. В нашем случае правильным ответом будет

```
Cross reference [NUL.CRF]: ex_prog
```

Помимо создания файла с расширением .CRF, указание имени в ответ на этот запрос заставляет Ассемблер включить номера строк в его обычный листинговый (.LST) файл.

После завершения трансляции введите команду

```
B>a:cref
```

и дайте следующие ответы на появляющиеся запросы:

```
Cref filename [.CRF]: ex_prog
List filename [EX_PROG.REF]: (нажмите клавишу возврата каретки)
```

Когда операционная система DOS вновь выдаст приглашение к вводу, изобразите на экране листинг перекрестных ссылок командой

```
B>type ex_prog.ref
```

Как показано на рис. 2.4, Вы получите имя каждого идентификатора программы, номер строки, где он определен (помеченный знаком #), и все номера строк других операторов, которые на него ссылаются.

```
EX-PROG - Пример программы

  Symbol Cross-Reference          (# is definition)          Cref-1

CODE . . . . . . . . . . . . . .  18
CSEG . . . . . . . . . . . . . .  18#   20    53

DATA . . . . . . . . . . . . . .   9
DEST . . . . . . . . . . . . . .  11#   36    37    38    39    44    46    48    50
DSEG . . . . . . . . . . . . . .   9#   15    20    31

OUR_PROG . . . . . . . . . . . .  19#   52    54

SOURCE . . . . . . . . . . . . .  10#   43    45    47    49
STACK. . . . . . . . . . . . . .   3#    3     8    20

 8 Symbols

62578 Bytes Free

A>
```

Рис. 2.4. Листинг перекрестных ссылок для примера программы

ЛИСТИНГ РАСПРЕДЕЛЕНИЯ ПАМЯТИ

Листинг распределения памяти содержит сводные сведения о сегментах программы. Для каждого сегмента указываются смещение адреса его начала (Start) и конца (Stop), длина сегмента в байтах и его категория (CODE – сегмент команд, DATA – сегмент данных, EXTRA – дополнительный сегмент и STACK – сегмент стека).

Листинг распределения памяти выдается загрузчиком LINK. Для этого его надо вызвать командой

```
B>a:link имя-программы,,;
```

(Обратите внимание на две запятые перед точкой с запятой.)

Для выдачи листинга распределения памяти на экран введите команду

```
B>type ex_prog.map
```

Если Вы хотите получить листинг на бумаге, то сначала наберите Ctrl-PrtSc.

Листинг должен выглядеть так, как показано на рис. 2.5. Когда его печать закончится, отмените печать появляющегося на экране текста, еще раз набрав Ctrl-PrtSc.

```
A>type ex-prog.map

 Start  Stop   Length Name                           Class
 00000H 001FFH 00200H STACK                          STACK
 00200H 00207H 00008H DSEG                           DATA
 00210H 00246H 00037H CSEG                           CODE

Program entry point at 0021:0000
```

Рис. 2.5. Листинг распределения памяти для примера программы

2.8. МОДЕЛИ СТРУКТУРЫ ПРОГРАММЫ

В этом разделе мы представим обобщенные модели, которыми можно пользоваться при повторении программ этой книги или при разработке собственных программ. Эти модели включают в себя все компоненты, которые должна иметь каждая программа. Вам нужно только заполнить их данными и командами, требующимися для Вашей программы.

Пример 2.1 является моделью исходного модуля, представляющего собой либо законченную программу, либо основной программный модуль, который должен загружаться вместе со вспомогательными модулями. Если подобный основной модуль должен содержать ссылки на элементы вспомогательных модулей, то в нем должен быть оператор EXTRN, перечисляющий эти элементы.

Пример 2.2 показывает модель вспомогательного модуля, который должен загружаться вместе с основным модулем примера 2.1. Обратите внимание на следующее:

1. Поскольку сегмент команд вспомогательного модуля имеет то же имя (CSEG), что и сегмент команд основного модуля, то процедура PNAME определена с атрибутом NEAR. Для вызова этой процедуры из сегмента команд, имеющего другое имя, определите процедуру PNAME с атирибутом FAR.
2. Поскольку, сегмент данных этого модуля имеет то же имя, что и сегмент данных основного модуля, то мы не инициализировали регистр DS. В этом нет нужды, так как команды основного модуля его уже инициализировали.

Однако в случае, если сегмент данных основного модуля имеет другое имя, надо вставить команды загрузки начального значения регистра DS и во вспомогательный модуль.

3. В примере процедура имеет условное имя PNAME. При написании своего модуля замените PNAME на имя своей процедуры в следующих трех местах: в операторе PUBLIC в начале модуля, в операторе PROC в его середине и операторе ENDP в конце модуля.

4. Данный модуль имеет оператор PUBLIC, соответствующий оператору EXTRN основного модуля. Если основной модуль содержит вызов процедуры PNAME, то он должен содержать оператор EXTRN PNAME:NEAR.

5. Псевдооператор END данного модуля не имеет метки, поскольку этот модуль не основной.

Вы можете создать эти модели так же, как и любую другую программу, используя EDLIN или какой-либо иной редактор, или программу обработки текста. Затем, когда Вам потребуется набрать текст программы, скопируйте соответствующую модель и дайте копии имя Вашей программы (например, copy mainmod.asm newprog.asm). Затем с помощью редактора вставьте в эту копию свои команды и данные.

ПРИМЕР 2.1. МОДЕЛЬ ОСНОВНОГО ПРОГРАММНОГО МОДУЛЯ

```
TITLE  (Разместите здесь заголовок)
          PAGE     ,132
(Если требуется оператор EXTRN, поместите его здесь)
STACK     SEGMENT   PARA STACK 'STACK'
         DB    64 DUP('STACK   ')    ;Область стека
STACK       ENDS
DSEG        SEGMENT   PARA PUBLIC 'DATA'

(Поместите здесь данные)

DSEG        ENDS
SUBTTL  Основная программа
            PAGE
CSEG        SEGMENT PARA PUBLIC 'CODE'
            ASSUME  CS:CSEG,DS:DSEG,SS:STACK

ENTRY       PROC    FAR          ;Точка входа

;  Занести в стек такие начальные значения, чтобы программа
;  могла возвратить управление отладчику DEBUG
;
            PUSH  DS
            SUB   AX,AX
            PUSH  AX

;
;  Инициировать адрес сегмента данных
;
            MOV   AX,DSEG
            MOV   DS,AX

(Поместите здесь команды)

            RET                  ;Возвратить управление DOS или DEBUG
ENTRY       ENDP
CSEG        ENDS
            END   ENTRY
```

ПРИМЕР 2.2. МОДЕЛЬ ВСПОМОГАТЕЛЬНОГО МОДУЛЯ

```
TITLE   (Разместите здесь заголовок)
          PAGE       ,132
          PUBLIC     PNAME
(Если требуется, поместите здесь оператор PUBLIC для переменных
сегмента данных)

DSEG      SEGMENT   PARA PUBLIC 'DATA'

(Поместите здесь данные)

DSEG      ENDS

CSEG      SEGMENT PARA PUBLIC 'CODE'
          ASSUME  CS:CSEG,DS:DSEG

PNAME     PROC     NEAR

(Поместите здесь команды)

          RET                ;Возвратиться в вызвавшую программу
PNAME     ENDP
CSEG      ENDS
          END
```

2.9. ДОПОЛНИТЕЛЬНЫЕ ПСЕВДООПЕРАТОРЫ

В разд. 2.5 мы обсуждали те псевдооператоры Ассемблера, которыми Вам придется пользоваться наиболее часто. В этом разделе описаны дополнительные, менее употребительные псевдооператоры или псевдооператоры, используемые более опытными программистами. В табл. 2.9 они разбиты на три группы: псевдооператоры данных, условные псевдооператоры и листинговые псевдооператоры.

ПСЕВДООПЕРАТОРЫ ДАННЫХ

Мы можем подразделить дополнительные псевдооператоры данных Ассемблера на *псевдооператоры определения блока* и *псевдооператоры управления трансляцией* (табл. 2.10).

Таблица 2.9. **Дополнительные псевдооператоры**

Тип	Псевдооператоры		
Псевдооператоры данных	EVEN GROUP	LABEL ORG	
Условные псевдооператоры	ELSE ENDIF IF IFDEF	IFNDEF IFDIF IFE IFIDN	IF1 IF2
Листинговые псевдооператоры	.CREF .LFCOND .LIST	%OUT .SFCOND .XCREF	.XLIST

Таблица 2.10. **Дополнительные псевдооператоры данных**

Псевдооператор	Функция
Определение блока	
GROUP	*Формат*: **имя** GROUP **имя-сег** [, . . .] Объединяет указанные сегменты в группу под одним именем так, чтобы они разместились в одном физическом сегменте объемом 64К.
LABEL	*Формат*: **имя** LABEL **тип** Задает атрибут для **имя**.
Управление трансляцией	
ORG	*Формат*: ORG **выражение** Полагает счетчик адреса равным значению **выражение**. Ассемблер присвоит этот адрес следующему объектному коду.
EVEN	*Формат*: EVEN Сдвигает значение счетчика адреса к ближайшему четному байту.

ПСЕВДООПЕРАТОРЫ ОПРЕДЕЛЕНИЯ БЛОКА

Псевдооператор GROUP (группа) собирает несколько сегментов в группу под одним именем так, чтобы поместить их в один блок памяти объемом в 64К. Этот псевдооператор Вам понадобится в том случае, если требуется разработать программу типа .COM (команда операционной системы), которая должна состоять из одного блока.

Например, если у Вашей программы сегмент команд имеет имя CODESEG, а сегмент данных – DATASEG, то их можно сгруппировать в один блок объемом 64К по имени CGROUP следующим образом:

```
CGROUP   GROUP    CODESEG,DATASEG
DATASEG  SEGMENT  PARA PUBLIC 'DATA'
         ..
         ..
DATASEG  ENDS
CODESEG  SEGMENT  PARA PUBLIC 'CODE'
         ASSUME   CS:CGROUP,DS:CGROUP
         ..
         ..
CODESEG  ENDS
         END
```

Псевдооператор LABEL определяет атрибуты *сегмента, смещения адреса* и *типа* заданного *имени*. Псевдооператором LABEL можно присвоить команде атрибут FAR и тем самым дать возможность команде перехода, находящейся в другом сегменте, передать ей управление. Например, операторы

```
HERE LABEL FAR
     MOV   DX,0
```

присваивают команде MOV метку HERE.

Псевдооператором LABEL можно воспользоваться для доступа к байтам в таблице слов или наоборот. Например, микропроцессор 8088 будет считать следую-

щую группу данных либо таблицей байтов по имени B_TABLE, либо таблицей слов по имени W_TABLE:

```
B_TABLE LABEL BYTE
W_TABLE DW 2F24H,36AH,0817H,3
```

ПСЕВДООПЕРАТОРЫ УПРАВЛЕНИЯ ТРАНСЛЯЦИЕЙ

Псевдооператор ORG (origin – начало) изменяет *счетчик адреса*, внутренний указатель, который сообщает Ассемблеру, в каком месте памяти надо хранить команды и данные. Обычно Вы оставляете решение о распределении памяти на усмотрение операционной системы DOS, но команда ORG дает Вам возможность принять это решение самому. Например, если Вы хотите использовать свою программу как файл типа .COM операционной системы DOS (см. руководство по DOS, в частности, описание команды EXE2BIN), то перед первой командой программы вставьте оператор

```
ORG 100H
```

Этот оператор сообщает Ассемблеру, что размещение команд программы в памяти надо начать, пропустив 256 байт от начала сегмента команд.

Псевдооператор EVEN (четный) используется довольно редко. С его помощью можно сделать более эффективным исполнение программ на ЭВМ, имеющих микропроцессор 8086 или 80286 (например, на персональной ЭВМ IBM AT). Имея 16-битовую шину данных, микропроцессор 8086 может передавать 16 битов информации за один прием (в отличие от микропроцессора 8088, которому для этого потребуется передать два раза по 8 битов). Однако при этом микропроцессор 8086 передает данные, начинающиеся с нечетных адресов памяти, дольше, чем данные, начинающиеся с четных адресов. Поэтому в приложениях, где время исполнения критично, важно иметь возможность запоминать данные в ячейках с четными адресами.

Чтобы обеспечить требуемое выравнивание размещения данных в памяти, расположите в своем сегменте данных вначале все двойные слова, затем слова, а последними – байты. Если сегмент содержит только псевдооператоры определения байтов, то укажите псевдооператор EVEN перед каждым из них, кроме первого. Например,

```
DSEG SEGMENT PARA PUBLIC 'DATA'
     HOURS DB ?
     EVEN
     MESSAGE DB 'Для продолжения нажмите любую клавишу'
DSEG ENDS
```

Если счетчик адреса содержит четное смещение, то псевдооператор EVEN никакого действия не вызовет, но если счетчик содержит нечетное смещение, то Ассемблер заменит псевдооператор EVEN на байт 00H с тем, чтобы сделать четным адрес следующей ячейки. Например, если счетчик адреса содержал смещение 129H, то псевдооператор EVEN заставит Ассемблер разместить следующий элемент данных по адресу 12AH.

УСЛОВНЫЕ ПСЕВДООПЕРАТОРЫ

Условные псевдооператоры заставляют Ассемблер либо транслировать, либо пропускать группу исходных операторов в зависимости от того, "истинно" или "ложно" в момент трансляции определенное условие. Эта избирательная

возможность "транслировать/не транслировать" позволяет Вам помещать в текст программы диагностические или специальные условия на случай тестовых прогонов или создавать специфические версии многоцелевой программы.

Для обеспечения условной трансляции порции текста программы вставьте перед ней псевдооператор IF (табл. 2.11), а после нее укажите псевдооператор ENDIF. Если условие в псевдооператоре IF окажется "истинным", то расположенные между IF и ENDIF операторы будут транслироваться; если же оно окажется "ложным", то эти операторы будут пропущены и трансляция продолжится со следующего после ENDIF оператора.

Мы можем сгруппировать восемь псевдооператоров IF в четыре пары следующим образом:

IFE дает значение "истинно", если **выражение** равно 0; IF дает значение "истинно", если **выражение** не равно 0.

IF1 дает значение "истинно", если Ассемблер выполняет первый проход; IF2 дает значение "истинно", если Ассемблер выполняет второй проход.

IFDEF дает значение "истинно", если **идентификатор** определен или объявлен как внешний псевдооператор EXTRN ; в противном случае значение "истинно" дает псевдооператор IFNDEF.

IFIDN дает значение "истинно", если строки **аргумент1** и **аргумент2** идентичны; IFIDF дает значение "истинно", если они различаются.

Таблица 2.11. **Условные псевдооператоры**

Псевдооператор	Значение
IFE	*Формат*: IFE **выражение** Истинно, если **выражение** равно нулю
IF	*Формат*: IF **выражение** Истинно, если **выражение** отлично от нуля
IF1	*Формат*: IF1 Истинно, если Ассемблер выполняет первый проход
IF2	*Формат*: IF2 Истинно, если Ассемблер выполняет второй проход
IFDEF	*Формат*: IFDEF **идентификатор** Истинно, если **идентификатор** определен или объявлен внешним в псевдооператоре EXTRN
IFNDEF	*Формат*: IFNDEF **идентификатор** Истинно, если **идентификатор** не определен и не объявлен внешним в псевдооператоре EXTRN
IFIDN	*Формат*: IFIDN <**строка1**>, <**строка2**> Истинно, если **строка1** и **строка2** идентичны. Угловые скобки необходимы.
IFIDF	*Формат*: IFIDF <**строка1**>, <**строка2**> Истинно, если **строка1** и **строка2** отличаются друг от друга. Угловые скобки необходимы.

Например, для включения диагностических процедур в тестовый прогон окаймите их псевдооператорами IFE и ENDIF и определите константу FOR_TEST_ONLY. Во время трансляции Ассемблер проверит ее значение; если оно окажется нулевым, то диагностические процедуры будут пропущены. Программа будет выглядеть следующим образом:

```
         IFE FOR_TEST_ONLY
DIAG1:    ... (Диагностические команды)
         ...
         ENDIF
```

Команды между меткой DIAG1 и оператором ENDIF будут оттранслированы только в том случае, если ранее в программу был помещен оператор

```
FOR_TEST_ONLY = 0
```

А оператор

```
FOR_TEST_ONLY = 1
```

вынудит Ассемблер пропустить команды, находящиеся между меткой DIAG1 и оператором ENDIF.

ПРИЗНАК АЛЬТЕРНАТИВЫ

Для включения в программу *альтернативной* группы команд в случае, если условие оказалось "ложным", можно воспользоваться признаком альтернативы ELSE (иначе). В общем случае он имеет следующий формат:

```
IFxx    [аргумент]
 ...    (Операторы для "истинного" значения условия)
 ...
[ELSE]
 ...    (Операторы для "ложного" значения условия)
 ...
ENDIF
```

Признак альтернативы ELSE позволяет, например, создать две версии программы, одна из которых будет выдавать приглашение к вводу и сообщения на английском языке, а другая – на испанском. Для этого можно определить константу LANGUAGE, которая позволит выбрать операторы, требуемые для соответствующего языка. Если ее значение равно 0, то Ассемблер создает английскую версию; если равно 1 – испанскую. В этом случае связанный с текстом сообщений раздел программы может иметь следующий вид:

```
IFE LANGUAGE
... (Операторы английской версии)
...
ELSE
... (Операторы испанской версии)
...
ENDIF
```

ВЛОЖЕННЫЕ УСЛОВНЫЕ ПСЕВДООПЕРАТОРЫ

Вы можете предложить для Ассемблера более двух вариантов трансляции с помощью *вложения* условных псевдооператоров. Предположим, что испанская версия Вашей программы действительно заработала и Вам нужно сделать другие версии, которые выдают сообщения и приглашения к вводу на

французском и немецком языках. Для этого модифицируйте программу так, чтобы Ассемблер выбирал язык по значениям 0, 1, 2 и 3 константы LANGUAGE (соответственно английский, испанский, французский или немецкий). В этом случае раздел сообщений может иметь следующий вид:

```
 IFE  LANGUAGE
   ..  (Операторы для выдачи сообщений на английском языке)
   ..
ELSE
   IFE  LANGUAGE-1
     ..  (Операторы для выдачи сообщений на испанском языке)
     ..
   ELSE
     IFE  LANGUAGE-2
     ..  (Операторы для выдачи сообщений на французском языке)
     ..
     ELSE
       ..  (Операторы для выдачи сообщений на немецком языке)
       ..
     ENDIF
   ENDIF
ENDIF
```

Обратите внимание, что нам потребовалось указать три отдельных оператора ENDIF для баланса с предыдущими операторами IFE. Обратите внимание и на то, что мы записали сочетание IFE-ENDIF с отступом. Ассемблеру этот отступ не требуется; мы сделали его для удобства чтения програмы.

ЛИСТИНГОВЫЕ ПСЕВДООПЕРАТОРЫ

Листинговые псевдооператоры указывают Ассемблеру, что печатать и в какой форме. Они сведены в табл. 2.12 в три функциональные группы.

Таблица 2.12. **Листинговые псевдооператоры**

Псевдооператор	Функция
Управление листингом	
.XCREF	*Формат*: .XCREF Отменяет листинг перекрестных ссылок вплоть до появления псевдооператора .CREF
.CREF	*Формат*: .CREF Возобновляет листинг перекрестных ссылок.
.XLIST	*Формат*: .XLIST Отменяет листинг программы вплоть до появления псевдооператора .LIST
.LIST	*Формат*: .LIST Возобновляет листинг программы.
Выдача сообщений во время трансляции	
%OUT	*Формат*: %OUT **текст** Изображает сообщение **текст**.

Псевдооператор	Функция
Управление листингом нетранслируемых блоков	
.LFCOND	*Формат*: .LFCOND Обеспечивает листинг всех условных блоков. Этот режим устанавливается по умолчанию.
.SFCOND	*Формат*: .SFCOND Исключает из листинга все нетранслируемые блоки.

ПСЕВДООПЕРАТОРЫ УПРАВЛЕНИЯ ЛИСТИНГОМ

Псевдооператоры .XCREF и .CREF позволяют исключить части программы из файла перекрестных ссылок, а псевдооператоры .XLIST и .LIST позволяют исключить их из файла с листингом программы (.LST). Вы можете использовать псевдооператоры .LIST и .XLIST для получения распечаток отдельных процедур длинной программы: вставляйте псевдооператор .LIST перед началом процедуры, а псевдооператор .XLIST – за ее концом.

ПСЕВДООПЕРАТОР ВЫДАЧИ СООБЩЕНИЯ ВО ВРЕМЯ ТРАНСЛЯЦИИ

В процессе трансляции программы псевдооператор %OUT изображает на экране заданное сообщение. Это полезно для выдачи сообщений о ходе трансляции длинной программы. Например, следующие операторы сообщат Вам о том, что Ассемблер завершил половину своего процесса двухпроходной трансляции:

```
IF2
  %OUT  Начинается второй проход трансляции
ENDIF
```

Если Вы увидели это сообщение, можно и подождать; в противном случае прогуляйтесь в кафетерий.

ПСЕВДООПЕРАТОРЫ УПРАВЛЕНИЯ ЛИСТИНГОМ НЕТРАНСЛИРУЕМЫХ БЛОКОВ

Из предыдущего обсуждения условных операторов Вы знаете, что если значение псевдооператора IF "ложно", то Ассемблер игнорирует все, что находится между операторами IF и ENDIF. Однако чтобы увидеть программу целиком, Вам может потребоваться распечатка этих нетранслируемых операторов.

Псевдооператоры .LFCOND и .SFCOND управляют листингом условных блоков. А именно, псевдооператор .LFCOND вызывает печать всего содержания условных блоков, а псевдооператор .SFCOND исключает из листинга нетранслируемые блоки.

2.10. ОБЗОР КЛЮЧЕВЫХ МОМЕНТОВ

Перечислим ключевые моменты, с которыми Вы познакомились в этой главе:

1. Строки программы, или *операторы*, могут представлять собой либо команды языка ассемблера, либо псевдооператоры. Команды языка ассемблера адресованы микропроцессору ЭВМ, а псевдооператоры – Ассемблеру.

2. Каждая команда языка ассемблера может иметь до четырех полей следующего вида:

```
[Метка:] Мнемокод [Операнд] [;Комментарий]
```

Поле метки содержит имя, присваиваемое команде. Оно позволяет другим командам ссылаться на нее. Каждая метка команды должна завершаться двоеточием. Поле мнемокода содержит от двух до шести букв, идентифицирующих команду. Поле операнда указывает микропроцессору 8088, где найти данные, подлежащие обработке. В поле комментариев можно дать краткое описание назначения команды; оно должно начинаться точкой с запятой.

3. Каждый псевдооператор может иметь до четырех полей следующего вида:

```
[Имя] Псевдооператор [Операнд] [;Комментарий]
```

4. Наиболее употребительные псевдооператоры можно разделить на две группы: псевдооператоры данных и псевдооператоры управления листингом.

5. *Псевдооператоры данных* можно подразделить на пять групп: псевдооператоры определения идентификаторов, псевдооператоры определения данных, псевдооператоры внешних ссылок, псевдооператоры определения сегмента/процедуры и псевдооператоры управления трансляцией.

6. Существуют два *псевдооператора определения идентификаторов*: EQU и = . Псевдооператор EQU присваивает имя выражению постоянно, а псевдооператор = – временно (так что это имя можно переопределить позже).

7. Существуют три *псевдооператора определения данных*: DB (определить байт), DW (определить слово) и DD (определить двойное слово). Обычно псевдооператоры DB и DW используются с целью резервирования ячеек памяти для переменных, в то время как с помощью оператора DD резервируют память для хранения адресов. Во всех трех случаях Вы можете либо указать начальное значение, либо просто зарезервировать ячейки памяти. Для резервирования ячеек памяти укажите в поле операнда вопросительный знак (?).

Для определения таблицы перечислите ее элементы через запятую. Чтобы Ассемблер повторил одно и то же значение несколько раз, используйте операцию DUP.

8. К *псевдооператорам определения сегмента/процедуры* относятся псевдооператоры SEGMENT и ENDS (для определения сегмента), PROC и ENDP (для определения процедуры), а также псевдооператор ASSUME (для идентификации вида определяемого Вами сегмента: сегмент данных, сегмент команд, дополнительный сегмент и сегмент стека).

Процедуры могут иметь атрибут NEAR или FAR. Процедуры с атрибутом NEAR могут быть вызваны только из того сегмента команд, в котором они определены, а процедуры с атрибутом FAR могут быть вызваны и из другого сегмента команд.

9. *Псевдооператоры внешних ссылок* позволяют Вам использовать объекты (например, процедуру или переменную), которые находятся в каком-то другом файле системы. Псевдооператор PUBLIC делает идентификаторы доступными другим модулям, которые, скорее всего, будут присоединяться к данному модулю в процессе загрузки. Псевдооператор EXTRN определяет внешние для данного модуля идентификаторы. Псевдооператор INCLUDE вставляет (на время трансляции) внешний файл в текущий исходный файл.

10. *Псевдооператор управления трансляцией* END отмечает конец исходного модуля, поэтому он должен присутствовать в каждом модуле.

11. *Псевдооператоры управления листингом* управляют формой выдаваемого Ассемблером листинга. Псевдооператор PAGE указывает длину и ширину страни-

цы, а псевдооператоры TITLE и SUBTTL могут быть использованы для выдачи заголовков и подзаголовков.

12. Существуют пять видов *операций*: арифметические операции; логические операции; операции отношения; операции, возвращающие значения, и операции присваивания атрибута.

13. *Арифметические операции* выполняют сложение, вычитание, умножение и целочисленное деление (+, –, * и /), а также операцию MOD, дающую остаток от деления нацело.

14. *Логические операции* AND, OR, XOR и NOT позволяют манипулировать отдельными битами двоичных чисел.

15. *Операции отношения* сравнивают значения двух операндов. Они позволяют проверять справедливость отношений "равно" (EQ), "не равно" (NE), "меньше" (LT), "больше" (GT), "меньше или равно" (LE) или "больше или равно" (GE).

16. Две наиболее употребительные *операции, возвращающие значения*, SEG и OFFSET – возвращают номер блока и значение смещения адреса переменной или метки.

17. Наиболее полезны *операции присваивания атрибута* DS:, ES:, SS: и CS:. Они позволяют задавать сегмент, отличный от того, который по умолчанию использует процессор при обмене данными с ячейкой памяти.

18. При разработке программы Вы должны ввести ее в ЭВМ (с помощью редактора текста EDLIN или другой программы обработки текста), затем оттранслировать, загрузить и вызвать ее для исполнения. Вы можете вызвать программу либо из отладчика DEBUG, либо из операционной системы DOS. В случае, если программа не выдает видимых результатов, пользуйтесь отладчиком DEBUG.

Пусть Ваш диск Ассемблера установлен в дисковод A, а диск данных – в дисковод B. Если активным является дисковод A, то перечисленные выше действия инициируются командами следующего вида:

```
a:edlin имя-прог.asm      (отредактировать)
a:masm имя-прог;          (оттранслировать)
a:link имя-прог;          (загрузить)
a:debug имя-прог.exe      (исполнить под управлением DEBUG)
имя-прог.exe              (исполнить под управлением DOS)
```

Для загрузки нескольких модулей перечислите их имена, соединенные знаком + (например, link mod1 + mod2;).

УПРАЖНЕНИЯ

1. Сколько *байтов* памяти зарезервирует следующая последовательность операторов:

```
VAR1   DB ?
VAR2   DW 4 DUP(?),20
VAR3   DB 10 DUP(?)
```

2. Какое значение будет помещено Ассемблером в переменную VAR1 из упр. 1?

3. Чем различаются следующие операторы:

```
K EQU 1024
K  =  1024
```

4. Какая ошибка содержится в операторах

```
CONST DB    ?
      MOV   CONST,256
```

5. Какими псевдооператорами отмечают начало и конец каждой процедуры?

6. Чем отличается процедура с атрибутом NEAR от процедуры с атрибутом FAR?

7. Почему процедура программы, которая исполняется микропроцессором 8088 первой, должна иметь атрибут FAR?

8. Что делает следующий оператор:

```
ASSUME CS:CSEG
```

9. Что делает загрузчик?

ГЛАВА 3. СИСТЕМА КОМАНД МИКРОПРОЦЕССОРА 8088

3.1. ОБ ЭТОЙ ГЛАВЕ

В гл. 2 Вы начали знакомиться с "базисом" языка ассемблера – командами, управляющими микропроцессором. В этой главе Вашему вниманию предлагается детальное описание системы команд микропроцессора 8088 (и 8086) и режимов адресации операндов.

Во многих книгах команды описаны по отдельности в алфавитном порядке. Хотя применительно к справочным техническим руководствам этот подход имеет определенные достоинства, но чтение таких книг утомляет и обескураживает уже после пятой или шестой команды.

В настоящей книге мы группируем команды по функциональному признаку, так что сходные команды описываются вместе. А именно, мы группируем команду сложения и команду вычитания, команды сдвига и команды циклического сдвига и т. д. Этот подход поможет Вам "понять" систему команд и "взаимоотношения" отдельных команд, а не выучить их.

Позже, после исполнения нескольких программ, Вам понадобится обращаться к данной главе лишь время от времени для выяснения деталей действия отдельных команд. Как только Вы в достаточной мере освоитесь с системой команд, то сможете получать ответы на большинство вопросов, заглядывая в приложение Г, где команды описаны в алфавитном порядке. Полезно и приложение В: в нем указано, сколько времени занимает исполнение каждой команды.

3.2. РЕЖИМЫ АДРЕСАЦИИ

Микропроцессор 8088 предоставляет Вам множество способов доступа к операндам, с которыми должна работать Ваша программа. Операнды могут содержаться в регистрах, в самих командах, в памяти или в портах ввода-вывода. В рекламных проспектах производителей оборудования утверждается, что микропроцессор 8088 имеет 24 режима адресации операндов. Пожалуй, так оно и есть, если рассматривать все возможные комбинации. В этой книге мы разделяем режимы адресации на семь групп:

1. Регистровая адресация.
2. Непосредственная адресация.
3. Прямая адресация.
4. Косвенная регистровая адресация.
5. Адресация по базе.
6. Прямая адресация с индексированием.
7. Адресация по базе с индексированием.

Микропроцессор выбирает *один* из семи режимов адресации по значению *поля режима* команды. Ассемблер присваивает то или иное значение *полю режима* в

зависимости от того, какой вид имеют операнды в исходной программе. Например, если Вы написали

```
MOV AX,BX
```

то Ассемблер закодирует оба операнда (AX и BX) для регистровой адресации. Однако если Вы заключили операнд-источник в квадратные скобки:

```
MOV AX,[BX]
```

то Ассемблер закодирует операнд-источник для *косвенной* регистровой адресации.

В табл. 3.1 приведены форматы операндов языка ассемблера для всех семи режимов адресации, реализуемых микропроцессором 8088, и для каждого формата указано, какой из регистров сегмента используется для вычисления физического адреса. Обратите внимание, что во всех режимах предполагается доступ к сегменту данных (т. е. регистром сегмента служит регистр DS), и только в тех случаях, когда используется регистр BP, предполагается доступ к сегменту стека (т. е. регистром сегмента служит регистр SS).

Важное замечание: при исполнении команд микропроцессора 8088, манипулирующих строками, предполагается, что регистр DI указывает на ячейку дополнительного сегмента, а не сегмента данных. Таким образом, в качестве регистра сегмента эти команды используют регистр ES. Все другие команды исполняются по правилам, описанным в табл. 3.1.

Таблица 3.1. **Режимы адресации микропроцессора 8088**

Режим адресации	Формат операнда	Регистр сегмента
Регистровый	**регистр**	Не используется
Непосредственный	**данное**	Не используется
Прямой	**сдвиг**	DS
	метка	DS
Косвенный регистровый	[BX]	DS
	[BP]	SS
	[DI]	DS
	[SI]	DS
По базе	[BX] + **сдвиг**	DS
	[BP] + **сдвиг**	CS
Прямой с индексированием	[DI] + **сдвиг**	DS
	[SI] + **сдвиг**	DS
По базе с индексированием	[BX] [SI] + **сдвиг**	DS
	[BX] [DI] + **сдвиг**	DS
	[BP] [SI] + **сдвиг**	SS
	[BP] [DI] + **сдвиг**	SS

Примечания:

1. Компонент **сдвиг** при адресации по базе с индексированием необязателен.
2. Операнд **регистр** может быть любым 8- или 16-битовым регистром, кроме регистра IP.
3. Операнд **данное** может быть 8- или 16-битовым значением константы.
4. Компонент **сдвиг** может быть 8- или 16-битовым значением смещения со знаком.

Из семи режимов адресации самыми быстрыми являются регистровая и непосредственная адресации операндов, поскольку в этом случае операционный блок микропроцессора 8088 извлекает их либо из регистров (при регистровой адресации), либо из конвейера команд (при непосредственной адресации). В других режимах адресация выполняется дольше, потому что интерфейс шины вначале должен вычислить адрес ячейки памяти, извлечь операнд и только после этого передать его операционному блоку.

Каждое описание режима адресации, приведенное в данном разделе, сопровождается примерами его применения. В большинстве случаев для этого используется команда MOV микропроцессора 8088.

РЕГИСТРОВАЯ И НЕПОСРЕДСТВЕННАЯ АДРЕСАЦИЯ

При *регистровой адресации* микропроцессор 8088 извлекает операнд из регистра (или загружает его в регистр). Например, команда

```
MOV AX,CX
```

копирует 16-битовое содержимое регистра счетчика CX в аккумулятор AX. Содержимое регистра CX не изменяется. В данном примере микропроцессор 8088 использует регистровую адресацию для извлечения операнда-источника из регистра CX и загрузки его в регистр-приемник AX.

Непосредственная адресация позволяет Вам указывать 8- или 16-битовое значение константы в качестве операнда-источника. Эта константа содержится в команде (куда она помещается Ассемблером), а не в регистре или в ячейке памяти. Например, команда

```
MOV CX,500
```

загружает значение 500 в регистр CX, а команда

```
MOV CL,-30
```

загружает значение –30 в регистр CL.

Непосредственный операнд может быть идентификатором, определенным оператором EQU, поэтому допустима следующая форма оператора:

```
K EQU 1024
   ...
   ...
   MOV CX,K
```

Чтобы избежать трудностей, помните, что допустимые значения для 8-битовых чисел со знаком ограничены диапазоном от –128 (80H) до 127 (7FH), а допустимые значения 16-битовых чисел со знаком – диапазоном от –32768 (8000H) до 32767 (7FFFH). Максимальные значения 8-битовых чисел без знака равны соответственно 255 (0FFH) и 65535 (0FFFFH).

РАСШИРЕНИЕ ЗНАКОВОГО БИТА НЕПОСРЕДСТВЕННЫХ ЗНАЧЕНИЙ

Ассемблер всегда расширяет знак при пересылке непосредственных значений в операнд-приемник. Это означает, что он дублирует старший значащий бит значения источника до тех пор, пока не будут заполнены все 8 или 16 битов операнда-приемника.

Например, операнд-источник нашего первого примера, десятичное число 500, может быть записано в виде 10-битового двоичного значения 0 111 110 100. Когда

Ассемблер устанавливает, что Вы требуете загрузить это значение в 16-битовый регистр CX, то он расширяет его до 16-битового, записав перед ним шесть копий "знакового" бита (со значением 0). Поэтому в регистр CX попадает двоичное значение 0 000 000 111 110 100. Во втором примере микропроцессор 8088 загружает в регистр CL 8-битовое двоичное представление 11 100 010 десятичного числа –30.

РЕЖИМЫ АДРЕСАЦИИ ПАМЯТИ

Как уже упоминалось в гл.1, доступ к ячейкам памяти обеспечивается взаимодействием операционного блока и интерфейса шины микропроцессора 8088. Когда операционному блоку требуется прочитать или записать значение операнда, находящегося в памяти, он передает значение смещения адреса интерфейсу шины. Последний добавляет это смещение к содержимому регистра сегмента (предварительно дополненному четырьмя нулями) и тем самым получает 20-битовый физический адрес, который и используется для доступа к операнду.

ИСПОЛНИТЕЛЬНЫЙ АДРЕС

Смещение, которое вычисляется операционным блоком для доступа к находящемуся в памяти операнду, называется *исполнительным адресом* операнда. Исполнительный адрес показывает, на каком расстоянии (в байтах) располагается операнд от начала сегмента, в котором он находится. Будучи 16-битовым числом без знака, исполнительный адрес позволяет получить доступ к операндам, находящимся выше начала сегмента на расстоянии до 65535 (или 64K) байтов.

Время, затрачиваемое операционным блоком на вычисление исполнительного адреса, является одним из основных компонентов общего времени исполнения команды. В зависимости от используемого режима адресации получение исполнительного адреса может заключаться всего лишь в извлечении его как составной части команды, но иногда могут потребоваться довольно долгие манипуляции, например сложение извлеченной из команды составляющей с регистром базы и с индексным регистром. Даже если время исполнения не является критичным для Вашей программы, стоит оценивать эти временные факторы в процессе чтения следующих ниже описаний режимов адресации.

ПРЯМАЯ АДРЕСАЦИЯ

При прямой адресации исполнительный адрес является составной частью команды (так же, как значения при непосредственной адресации). Микропроцессор 8088 добавляет этот исполнительный адрес к сдвинутому содержимому регистра сегмента данных DS и получает 20-битовый физический адрес операнда.

Обычно прямая адресация применяется, если операндом служит метка. Например, команда

```
MOV AX,TABLE
```

загружает содержимое ячейки памяти TABLE в регистр AX. На рис. 3.1 показана схема исполнения этой команды. Обратите внимание на то, что против ожидания микропроцессор 8088 заполняет данные в памяти в обратном порядке. Старший байт слова *следует* за младшим байтом, а не предшествует ему. Чтобы усвоить это, запомните, что *старшая часть (старшие биты)* данных *располагается в ячейках памяти со старшими адресами.*

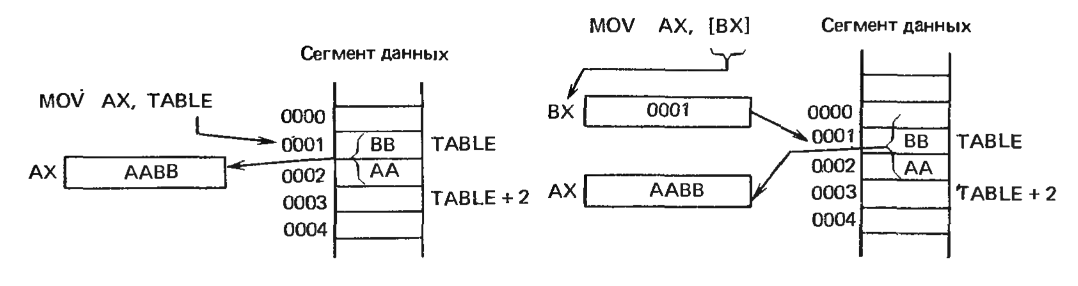


Рис. 3.1. Прямая адресация

Рис. 3.2. Косвенная регистровая адресация

КОСВЕННАЯ РЕГИСТРОВАЯ АДРЕСАЦИЯ

При косвенной регистровой адресации исполнительный адрес операнда содержится в базовом регистре BX, регистре указателя базы BP или индексном регистре (SI или DI). Косвенные регистровые операнды надо заключать в квадратные скобки, чтобы отличить их от регистровых операндов. Например, команда

```
MOV AX,[BX]
```

загружает в регистр AX содержимое ячейки памяти, адресуемой значением регистра BX (рис. 3.2).

Как поместить смещение адреса в регистр BX? Один из методов состоит в применении операции OFFSET (смещение) к адресу ячейки памяти. Например, для загрузки слова из ячейки TABLE в регистр AX можно воспользоваться последовательностью команд

```
MOV BX,OFFSET TABLE
MOV AX,[BX]
```

Эти две команды выполняют те же действия, что и одна команда

```
MOV AX,TABLE
```

с той лишь разницей, что в первом случае предыдущее содержимое регистра BX уничтожается. Если Вам нужен доступ лишь к одной ячейке памяти (в данном случае TABLE), то разумнее воспользоваться одной командой. Однако для доступа к *нескольким* ячейкам, начиная с данного базового адреса, гораздо лучше иметь исполнительный адрес в регистре. Почему? Потому что содержимым регистра можно манипулировать, не извлекая каждый раз новый адрес.

АДРЕСАЦИЯ ПО БАЗЕ

При адресации по базе Ассемблер вычисляет исполнительный адрес с помощью сложения значения сдвига с содержимым регистров BX или BP.

Регистр BX удобно использовать при доступе к структурированным записям данных, расположенным в разных областях памяти. В этом случае базовый адрес записи помещается в базовый регистр BX и доступ к ее отдельным элементам осуществляется по их сдвигу относительно базы. А для доступа к разным записям одной и той же структуры достаточно соответствующим образом изменить содержимое базового регистра.

Предположим, например, что требуется прочитать с диска учетные записи для ряда работников. При этом каждая запись содержит табельный номер работника, номер отдела, номер группы, возраст, тарифную ставку и т.д. Если номер отдела

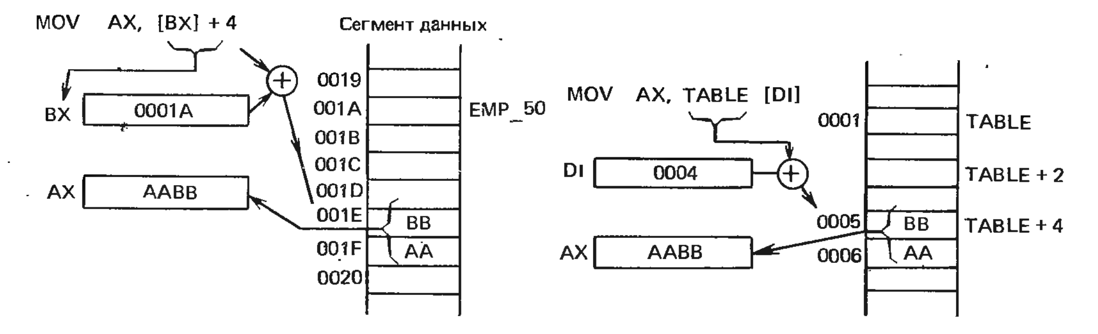


Рис. 3.3. Адресация по базе

Рис. 3.4. Прямая адресация с индексированием

хранится в пятом и шестом байтах записи, а начальный адрес записи содержится в регистре BX, то команда

```
MOV AX,[BX]+4
```

загрузит в регистр AX номер отдела, в котором служит данный работник (рис. 3.3). (Сдвиг равен 4, а не 5, потому что первый байт записи имеет номер 0.)

Ассемблер фирмы IBM позволяет указывать адресуемые по базе операнды тремя разными способами. Следующие команды эквивалентны:

```
MOV AX, [BP]+4       ;Это стандартная форма записи,
MOV AX, 4[BP]        ; но сдвиг можно указать на первом месте
MOV AX, [BP+4]       ; или внутри скобок
```

ПРЯМАЯ АДРЕСАЦИЯ С ИНДЕКСИРОВАНИЕМ

При прямой адресации с индексированием исполнительный адрес вычисляется как сумма значений сдвига и индексного регистра (DI или SI). Этот тип адресации удобен для доступа к элементам таблицы, когда сдвиг указывает на начало таблицы, а индексный регистр – на ее элемент.

Например, если B_TABLE – таблица байтов, то последовательность команд

```
MOV DI,2
MOV AL,B_TABLE[DI]
```

загрузит третий элемент таблицы в регистр AL.

В таблице слов соседние элементы отстоят друг от друга на два байта, поэтому при работе с ней надо *удваивать номер элемента* при вычислении значения индекса. Если TABLE – таблица слов, то для загрузки в регистр AX ее третьего элемента надо использовать последовательность команд

```
MOV DI,4
MOV AX,TABLE[DI]
```

(рис. 3.4).

АДРЕСАЦИЯ ПО БАЗЕ С ИНДЕКСИРОВАНИЕМ

При адресации по базе с индексированием исполнительный адрес вычисляется как сумма значений базового регистра, индексного регистра и, возможно, сдвига.

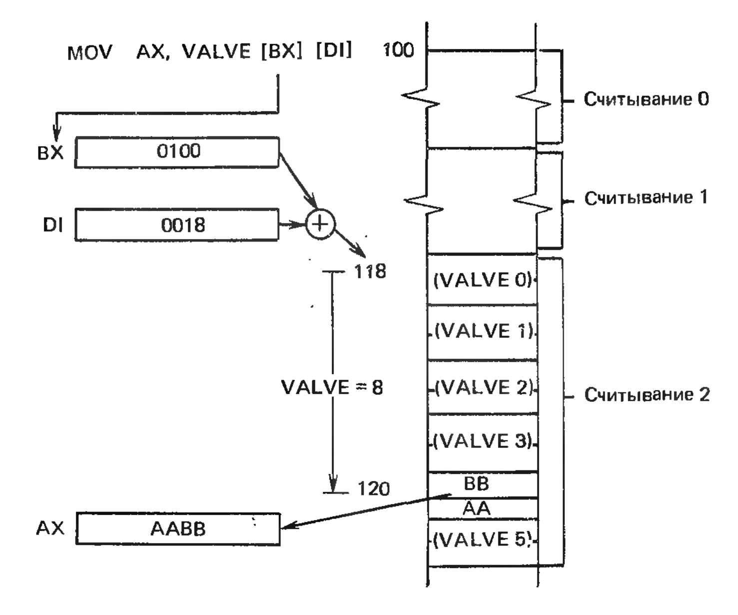


Рис. 3.5. Извлечение значения элемента двумерного массива

Так как в этом режиме адресации складывается два отдельных смещения, то он удобен при адресации двумерных массивов, когда базовый регистр содержит начальный адрес массива, а значения сдвига и индексного регистра суть смещения по строке и столбцу.

Предположим, например, что Ваша ЭВМ следит за шестью предохранительными клапанами на химическом предприятии. Она считывает их состояния каждые полчаса и запоминает в ячейках памяти. За неделю эти считывания образуют массив, состоящий из 336 блоков (48 считываний в течение семи дней) по шесть элементов в каждом, а всего – 2016 значений.

Если начальный адрес массива загружен в регистр BX, сдвиг блока (номер считывания, умноженный на 12) – в регистре DI, а номер клапана задан в переменной VALVE, то команда

```
MOV AX,VALVE[BX][DI]
```

загрузит требуемое считывание состояния клапана в регистр AX. На рис. 3.5 изображен процесс извлечения результата третьего считывания (с номером 2) для клапана 4 из массива, у которого смещение в сегменте данных равно 100H.

Приведем несколько допустимых форматов операндов, адресуемых по базе с индексированием:

```
MOVE AX,[BX+2+DI]        ;Операнды можно заключать в скобки в любом
MOVE AX,[DI+BX+2]        ; порядке, а сдвиг можно сочетать с любым
MOVE AX,[BX+2][DI]       ; из регистров
MOVE AX,[BX][DI+2]       ;
```

3.3. ТИПЫ КОМАНД

Как уже упоминалось, микропроцессор 8088 имеет 92 типа команд. В табл. 3.2 приведены их мнемокоды на языке ассемблера и кратко указано назначение. Учтите, что некоторые команды имеют по нескольку разных мнемокодов.

Таблица 3.2. Система команд микропроцессора 8088

Мнемокод	Назначение
AAA	Скорректировать сложение для представления в кодах ASCII
AAD	Скорректировать деление для представления в кодах ASCII
AAM	Скорректировать умножение для представления в кодах ASCII
AAS	Скорректировать вычитание для представления в кодах ASCII
ADC	Сложить с переносом
ADD	Сложить
AND	Выполнить операцию И
CALL	Вызвать процедуру
CBW	Преобразовать байт в слово
CLC	Обнулить флаг переноса
CLD	Обнулить флаг направления
CLI	Обнулить флаг прерывания
CMC	Обратить флаг переноса
CMP	Сравнить значения
CMPS, CMPSB или CMPSW	Сравнить строки
CWD	Преобразовать слово в двойное слово
DAA	Скорректировать сложение для представления в десятичной форме
DAS	Скорректировать вычитание для представления в десятичной форме
DEC	Уменьшить значение
DIV	Поделить
ESC	Передать команду сопроцессору
HLT	Остановиться
IDIV	Разделить целые числа
IMUL	Умножить целые числа
IN	Считать значение из порта
INC	Прирастить значение
INT	Прервать
INTO	Прервать при переполнении
IRET	Возвратиться после прерывания

Мнемокод	Назначение
JA или JNBE	Перейти, если выше
JAE, JNB или	Перейти, если выше или равно
JNC	Перейти, если нет переноса
JB, JNAE или	Перейти, если ниже
JC	Перейти, если перенос
JBE или JNA	Перейти, если ниже или равно
JCXZ	Перейти, если содержимое регистра CX равно нулю
JE или JZ	Перейти, если равно
JG или JNLE	Перейти, если больше
JGE или JNL	Перейти, если больше или равно
JL или JNGE	Перейти, если меньше
JLE или JNG	Перейти, если меньше или равно
JMP	Перейти безусловно
JNE или JNZ	Перейти, если не равно
JNO	Перейти, если нет переполнения
JNP или JPO	Перейти, если нет четности
JNS	Перейти, если знаковый разряд нулевой
JO	Перейти, если переполнение
JP или JPE	Перейти, если есть четность
JS	Перейти, если знаковый разряд равен 1
LAHF	Загрузить регистр AH флагами
LDS	Загрузить указатель с использованием регистра DS
LEA	Загрузить исполнительный адрес
LES	Загрузить указатель с использованием регистра ES
LOCK	Замкнуть шину
LODS, LODSB или LODSW	Загрузить строку
LOOP	Повторять цикл до конца счетчика
LOOPE или LOOPZ	Повторять цикл, если равно

Мнемокод	Назначение
LOOPNE или LOOPNZ	Повторять цикл, если не равно
MOV	Переслать значение
MOVS, MOVSB или MOVSW	Переслать строку
MUL	Умножить
NEG	Обратить знак
NOP	Нет операции
NOT	Обратить биты
OR	Выполнить операцию ИЛИ
OUT	Вывести значение в порт
POP	Поместить значение в стек
POPF	Поместить флаги в стек
PUSH	Извлечь значение из стека
PUSHF	Извлечь флаги из стека
RCL	Сдвинуть влево циклически с флагом переноса
RCR	Сдвинуть вправо циклически с флагом переноса
REP, REPE или REPZ	Повторять, пока равно
REPNE или REPNZ	Повторять, пока не равно
RET	Возвратиться в вызывающую процедуру
ROL	Сдвинуть влево циклически
ROR	Сдвинуть вправо циклически
SAHF	Загрузить флаги из регистра AH
SAL или SHL	Сдвинуть влево арифметически
SAR	Сдвинуть вправо арифметически
SBB	Вычесть с заемом
SCAS, SCASB или SCASW	Сканировать строку
SHR	Сдвинуть вправо логически
STC	Установить флаг переноса

Мнемокод	Назначение
STD	Установить флаг направления
STI	Установить флаг прерывания
STOS, STOSB или STOSW	Сохранить строку
SUB	Вычесть
TEST	Проверить
WAIT	Ожидать
XCHG	Обменять значения
XLAT	Выбрать значения из таблицы
XOR	Выполнить операцию ИСКЛЮЧАЮЩЕЕ ИЛИ

Мы можем разделить систему команд на семь функциональных групп:

1. *Команды пересылки данных*, осуществляющие обмен информацией между регистрами, ячейками данных и портами ввода-вывода.

2. *Арифметические команды*, выполняющие арифметические операции над двоичными или двоично-десятичными (в формате BCD – binary-coded decimal) числами.

3. *Команды манипулирования битами*, выполняющие сдвиг, циклический сдвиг и логические операции со значениями регистров и ячеек памяти.

4. *Команды передачи управления*, управляющие последовательностью исполнения команд программы. К ним относятся переходы к другой команде, вызов процедуры и возврат из нее.

5. *Команды обработки строк*, перемещающие, сравнивающие и сканирующие строки данных.

6. *Команды прерывания*, отвлекающие микропроцессор на обработку некоторых специфичных ситуаций.

7. *Команды управления процессором*, устанавливающие и сбрасывающие флаги состояния, а также изменяющие режим функционирования микропроцессора.

В следующих разделах мы опишем систему команд микропроцессора 8088 по группам в перечисленном выше порядке. Начнем с группы команд пересылки данных, которая содержит вездесущую команду MOV.

3.4. КОМАНДЫ ПЕРЕСЫЛКИ ДАННЫХ

Команды пересылки данных осуществляют обмен данными и адресами между регистрами и ячейками памяти или портами ввода-вывода. В табл. 3.3 эти команды разделены на четыре подгруппы: команды общего назначения, команды ввода-вывода, команды пересылки адреса и команды пересылки флагов.

Таблица 3.3. **Команды пересылки данных**

Мнемокод	Формат	Флаги								
		OF	DF	IF	TF	SF	ZF	AF	PF	CF
Команды общего назначения										
MOV	MOV **приемник, источник**	–	–	–	–	–	–	–	–	–
PUSH	PUSH **источник**	–	–	–	–	–	–	–	–	–
POP	POP **приемник**	–	–	–	–	–	–	–	–	–
XCHG	XCHG **приемник, источник**	–	–	–	–	–	–	–	–	–
XLAT	XLAT **таблица_источник**	–	–	–	–	–	–	–	–	–
Команды ввода-вывода										
IN	IN **аккумулятор, порт**	–	–	–	–	–	–	–	–	–
OUT	OUT **порт, аккумулятор**	–	–	–	–	–	–	–	–	–
Команды пересылки адреса										
LEA	LEA **регистр 16, память 16**	–	–	–	–	–	–	–	–	–
LDS	LDS **регистр 16, память 32**	–	–	–	–	–	–	–	–	–
LES	LES **регистр 16, память 32**	–	–	–	–	–	–	–	–	–
Команды пересылки флагов										
LAHF	LAHF	–	–	–	–	–	–	–	–	–
SAHF	SAHF	–	–	–	–	*	*	*	*	*
PUSHF	PUSHF	–	–	–	–	–	–	–	–	–
POPF	POPF	*	*	*	*	*	*	*	*	*

Примечание. * означает изменение значения флага, — означает его сохранение.

КОМАНДЫ ОБЩЕГО НАЗНАЧЕНИЯ

КОМАНДА MOV

Основная команда общего назначения MOV (move – переслать) может пересылать байт или слово между регистром и ячейкой памяти или между двумя регистрами. Она может также пересылать непосредственно адресуемое значение в регистр или в ячейку памяти.

Команда MOV имеет следующий формат:

```
MOV приемник,источник
```

В ней допустимо большинство из возможных сочетаний операндов.

Приведем несколько примеров:

```
MOV AX,TABLE       ;Пересылка из памяти в регистр
MOV TABLE,AX       ; и наоборот
MOV ES:[BX],AX     ;Можно заменить используемый регистр сегмента
MOV DS,AX          ;Пересылка между 16-битовыми регистрами
MOV BL,AL          ;Пересылка между 8-битовыми регистрами
MOV CL,-30         ;Пересылка константы в регистр
MOV DEST,25H       ; или в память
```

В команде MOV *исключаются* следующие сочетания операндов:

1. Вы не можете осуществить непосредственную пересылку данных из одной ячейки памяти в другую. Чтобы выполнить такую пересылку, данные источника надо загрузить в регистр общего назначения, а затем запомнить содержимое *этого* регистра в приемнике. Например, если POUNDS и WEIGHT – переменные, находящиеся в памяти, то для пересылки значения из одной переменной в другую можно воспользоваться командами

```
MOV  AX,POUNDS
MOV  WEIGHT,AX
```

2. Вы не можете загрузить непосредственно адресуемый операнд в регистр сегмента. Как и в случае 1, сначала надо загрузить его в регистр общего назначения. Например, следующие команды загружают номер блока сегмента данных (DATA_SEG) в регистр DS:

```
MOV AX,DATA_SEG
MOV DS,AX
```

Подобные команды обычно сопутствуют оператору ASSUME в сегменте команд. Они указывают Ассемблеру, где размещен сегмент данных.

3. Вы не можете непосредственно переслать значение одного регистра сегмента в другой. Делайте подобные пересылки через регистр общего назначения. Например, чтобы регистр DS указывал на тот же сегмент, что и регистр ES, воспользуйтесь командами

```
MOV AX,ES
MOV DS,AX
```

(Для выполнения этой операции можно воспользоваться командами PUSH и POP, которые будут описаны в следующем разделе.)

4. Вы не можете использовать регистр CS в качестве приемника в команде пересылки.

КОМАНДЫ PUSH И POP

Как уже упоминалось, во время исполнения процедуры стек содержит адрес возврата. Команда вызова процедуры CALL (call – вызвать) помещает адрес в стек, а команда возврата RET (return – возвратить) извлекает его по окончании исполнения процедуры. Это один из случаев, когда микропроцессор 8088 использует стек *автоматически*, без Вашего на то указания.

Таким образом, стек удобен для временного сохранения данных (содержимого регистров и ячеек памяти) при работе Вашей программы. Например, Вам может понадобиться сохранить содержимое регистра AX на то время, пока он требуется для выполнения каких-либо действий. В Вашем распоряжении имеются две команды для работы со стеком – PUSH (поместить слово в стек) и POP (извлечь слово из стека).

Команда PUSH помещает содержимое регистра или ячейки памяти размером в 16-битовое слово на вершину стека. А команда POP, наоборот, снимает слово с вершины стека и помещает его в ячейку памяти или регистр.

Команды PUSH и POP имеют следующие форматы:

```
PUSH источник
POP  приемник
```

Приведем несколько примеров:

```
PUSH   SI                ;Вы можете сохранить регистр общего назначения
PUSH   DS                ; или регистр сегмента,
PUSH   CS                ; включая регистр CS
PUSH   COUNTER           ;Вы можете также сохранить содержимое
PUSH   TABLE[BX][DI]     ; ячейки памяти
```

Будучи взаимно обратными командами, PUSH и POP обычно используются парами, т.е. каждой команде PUSH в программе должна соответствовать своя команда POP. Например, при сохранении содержимого регистра AX в стеке и последующем его восстановлении Ваша программа будет иметь вид

```
PUSH AX          ;Сохранить AX на вершине стека
...              (Другие операции, изменяющие содержимое AX)
...
POP AX           ;Снять значение AX с вершины стека
```

Под *вершиной* стека мы понимаем ячейку в сегменте стека, адрес которой содержится в указателе стека SP. Так как стек "растет" по направлению к младшим адресам памяти (к ячейке 0), то первое помещаемое в стек слово запоминается в ячейке стека с наибольшим адресом, следующее – на два байта ниже и т.д.

Регистр SP всегда указывает на слово, помещенное в стек последним. Следовательно, команда PUSH вычитает 2 из значения указателя стека, а затем пересылает операнд-источник (слово) в стек. Действуя обратным образом, команда POP пересылает в операнд-приемник слово, адрес которого содержится в регистре SP, а затем добавляет 2 к содержимому этого регистра. На рис 3.6 показаны состояния

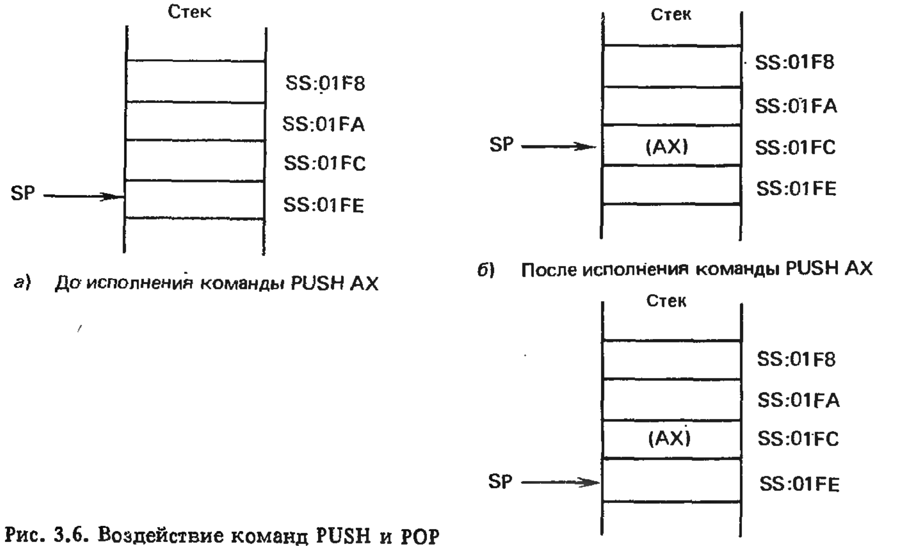


Рис. 3.6. Воздействие команд PUSH и POP на стек

стека и его указателя до и после использования команд PUSH и POP. В результате исполнения команды PUSH (рис. 3.6, *б*) указатель стека передвинется на два байта памяти ниже, и в этих байтах (ранее не используемых) будет содержаться значение регистра AX. В результате исполнения команды POP (рис. 3.6, *в*) содержимое регистра SP вернется в исходное состояние. (Обратите внимание на то, что хотя содержимое регистра AX по-прежнему находится в памяти, оно не принадлежит стеку.)

С помощью серии команд PUSH в стеке можно сохранить более одного слова. Однако помните, что каждая команда PUSH помещает свой операнд на вершину стека, и поэтому *с помощью команды POP Вы будете извлекать слова в порядке, обратном их помещению.* Следующая последовательность команд помещает значения четырех регистров в стек, а затем восстанавливает их:

```
PUSH  AX       ;Сохранить AX,
PUSH  ES       ; ES,
PUSH  DI       ;  DI и
PUSH  SI       ;   SI
...
...
POP   SI       ;Восстановить SI,
POP   DI       ; DI,
POP   ES       ;  ES и
POP   AX       ;   AX
```

Команды PUSH и POP удобны также для копирования содержимого одного регистра сегмента в другой. Например, с помощью команд

```
PUSH  ES
POP   DS
```

можно скопировать значение регистра ES в регистр DS.

Этот метод обладает тем преимуществом, что при его применении не надо использовать регистр общего назначения для промежуточного копирования значения регистра сегмента (см. предыдущий подраздел). А его недостаток заключается в том, что исполнение пары команд PUSH-POP занимает 26 циклов тактового генератора, в то время как пара команд MOV исполняется всего за 4 цикла.

Для помещения в стек и извлечения из него содержимого флагов имеются специальные команды. Мы рассмотрим их в подразделе "Команды пересылки флагов".

КОМАНДА ОБМЕНА XCHG

Команда обмена XCHG (exchange – обменять) меняет между собой значения двух регистров или регистра и ячейки памяти. Однако она не может выполнить обмен значений регистров сегмента.

Приведем несколько примеров команд XCHG:

```
XCHG  AX,BX            ;Обменять значения двух регистров (слова
XCHG  AL,BH            ; или байты)
XCHG WORD_LOC,DX       ;Обменять значения ячейки памяти
XCHG DL,BYTE_LOC       ; и регистра
```

КОМАНДА ИЗВЛЕЧЕНИЯ ЭЛЕМЕНТА ТАБЛИЦЫ XLAT

Команда извлечения элемента таблицы XLAT (translate – переводить с одного языка на другой) выбирает значение из таблицы байтов и загружает его в регистр AL. Таблица может иметь до 256 элементов.

Команда XLAT имеет формат

```
XLAT таблица_источник
```

где **таблица_источник** – имя просматриваемой таблицы. Перед исполнением команды XLAT начальный адрес таблицы надо загрузить в регистр BX, а номер извлекаемого байта – в регистр AL.

Следующая последовательность команд извлекает десятый байт из таблицы S_TAB:

```
MOV AL,10                   ;Загрузить номер байта в AL
MOV BX,OFFSET S_TAB         ;Загрузить смещение адреса в BX
XLAT S_TAB                  ;Извлечь значение байта из таблицы в AL
```

Команда XLAT удобна для выполнения преобразований, которые требуют многочисленных действий, например для поиска ASCII-кода шестнадцатеричной цифры.

КОМАНДЫ ВВОДА-ВЫВОДА

Команды ввода-вывода используются для взаимодействия с периферийными устройствами системы. Они имеют формат

```
IN аккумулятор,порт
OUT порт,аккумулятор
```

где **аккумулятор** – регистр AL при обмене байтами или регистр AX при обмене словами. Операндом **порт** может быть десятичное значение от 0 до 255, что позволяет адресоваться к 256 устройствам.

В качестве операнда **порт** можно использовать регистр DX, что позволяет легко изменять номер порта, например при необходимости пересылать одни и те же данные в несколько различных портов.

Приведем несколько примеров команд IN и OUT:

```
IN  AL,200             ;Ввести байт из порта 200
IN  AL,PORT_VAL        ; или из порта, указанного константой
OUT 30H,AX             ;Вывести слово в порт 30H
OUT DX,AX              ; или в порт, указанный в DX
```

КОМАНДЫ ПЕРЕСЫЛКИ АДРЕСА

Команды пересылки адреса передают не *содержимое* переменных, а их *адреса*.

КОМАНДА ЗАГРУЗКИ ИСПОЛНИТЕЛЬНОГО АДРЕСА LEA

Команда LEA (load effective address – загрузить исполнительный адрес) пересылает смещение ячейки памяти в любой 16-битовый регистр общего назначения, регистр указателя или индексный регистр. Она имеет формат

```
LEA регистр16,память16
```

где операнд **память16** должен иметь атрибут типа WORD.

В отличие от команды MOV с операцией OFFSET, операнд **память16** в команде LEA может быть индексирован, что дает возможность осуществить гибкую адресацию. Например, если регистр DI содержит 5, то команда

```
LEA BX,TABLE[DI]
```

загрузит смещение адреса TABLE+5 в регистр BX.

Мы более детально обсудим команду LEA в разд. 3.8 при описании операций над строками.

КОМАНДА ЗАГРУЗКИ УКАЗАТЕЛЯ И РЕГИСТРА СЕГМЕНТА ДАННЫХ LDS

Команда LDS (load pointer using DS – загрузить указатель с использованием регистра DS) считывает из памяти 32-битовое двойное слово и загружает первые 16 битов в заданный регистр, а следующие 16 битов – в регистр сегмента данных DS. Она имеет формат

```
LDS регистр16,память32
```

где **регистр16** – любой 16-битовый регистр общего назначения, а **память32** – ячейка памяти с атрибутом типа DOUBLEWORD.

Обычно операнд **память32** определяется псевдооператором DD (Define Doubleword – определить двойное слово), обсуждавшимся в разд. 2.5. Используя приведенный в этом разделе пример

```
HERE_FAR  DD  HERE
```

мы можем поместить смещение и номер блока адреса метки HERE в регистры BX и DS соответственно с помощью команды

```
LDS BX,HERE_FAR
```

Таким образом, одна команда LDS заменяет группу команд пересылки

```
MOV BX,OFFSET HERE
MOV AX,SEG HERE
MOV DS,AX
```

Обратите внимание на то, что применение команды LDX исключает необходимость в использовании третьего регистра (в нашем примере регистра AX).

КОМАНДА ЗАГРУЗКИ УКАЗАТЕЛЯ И РЕГИСТРА ДОПОЛНИТЕЛЬНОГО СЕГМЕНТА LES

Команда LES (load pointer using ES – загрузить указатель с использованием регистра ES) идентична команде LDS, но загружает номер блока в регистр ES, а не в DS.

КОМАНДЫ ПЕРЕСЫЛКИ ФЛАГОВ

КОМАНДА ЗАГРУЗКИ РЕГИСТРА AH ФЛАГАМИ (LAHF) И КОМАНДА ЗАГРУЗКИ ФЛАГОВ ИЗ РЕГИСТРА AH (SAHF)

Команда LAHF (Load AH from Flags – загрузить регистр AH флагами) копирует флаги, совместимые с флагами микропроцессоров 8080/8085, в регистр AH. А именно, она копирует флаги CF, PF, AF, ZF и SF в соответствующие биты регистра AH (0, 2, 4, 6 и 7). Команда SAHF (Store AH into Flags – загрузить флаги из регистра AH) выполняет обратную операцию: она загружает пять упомянутых выше битов регистра AH в регистр флагов.

Команда LAHF не изменяет состояния флагов. Команда SAHF, конечно же, изменяет состояние флагов микропроцессоров 8080/8085. Эти команды введены для совместимости с микропроцессорами 8080/8085.

КОМАНДЫ ПОМЕЩЕНИЯ ФЛАГОВ В СТЕК PUSHF И ИЗВЛЕЧЕНИЯ ФЛАГОВ ИЗ СТЕКА POPF

Эти команды пересылают содержимое регистра флагов в стек и обратно. Они в сущности идентичны командам PUSH и POP, но в них не требуется указывать операнд, так как под ним подразумевается регистр флагов.

Как и в случае команд PUSH и POP, команды PUSHF и POPF всегда используются парами. Другими словами, каждой команде PUSHF должна соответствовать исполняемая позже команда POP, например

```
PUSHF           ;Сохранить флаги в стеке
...              (Выполнить другие команды,
...              изменяющие состояние флагов)
POPF            ;Восстановить флаги из стека
```

Учтите, что с помощью команд PUSH, PUSHF, POP и POPF можно сохранить содержимое любого регистра (или даже всех регистров) на время исполнения процедуры или программы обработки прерывания. Пусть, например, содержащиеся в регистрах AX, DI и SI данные имеют существенное значение, а Вам требуется обратиться к процедуре SORT, которая может изменить их содержимое. Предположим также, что только что была выполнена арифметическая операция и важно оставить флаги неприкосновенными. Эту задачу выполнит следующая последовательность команд:

```
PUSH AX             ;Сохранить три регистра
PUSH DI
PUSH SI
PUSHF               ; и флаги
CALL SORT           ;Вызвать процедуру
POPF                ;По возвращению восстановить флаги
POP SI              ; и три регистра
POP DI
POP AX
```

Но лучше всего включить требуемые команды PUSH, PUSHF и POP, POPF *в тело* процедуры с тем, чтобы не повторять их при каждом обращении к ней. Следовательно, четыре команды PUSH из предыдущего примера должны стать первыми командами процедуры SORT, а четыре команды POP – ее последними командами. После этого можно указывать команду CALL SORT, не думая о том, какие регистры уничтожаются, а какие сохраняются.

3.5. АРИФМЕТИЧЕСКИЕ КОМАНДЫ

Микропроцессор 8088 может выполнять арифметические команды над двоичными числами со знаком или без знака, а также над десятичными числами без знака (как упакованными, так и неупакованными). Как показано в табл. 3.4, имеются команды для выполнения четырех стандартных действий арифметики – сложения, вычитания, умножения и деления, а также две дополнительные команды для действий над операндами с *расширенным* знаком, позволяющие оперировать со смешанными данными. Прежде чем приступить к описанию самих команд, рассмотрим те виды данных, с которыми они работают.

Таблица 3.4. **Арифметические команды**

Мнемокод	Формат	Флаги								
		OF	DF	IF	TF	SF	ZF	AF	PF	CF
Команды сложения										
ADD	ADD приемник, источник	*	—	—	—	*	*	*	*	*
ADC	ADC приемник, источник	*	—	—	—	*	*	*	*	*
AAA	AAA	?	—	—	—	?	?	*	?	*
DAA	DAA	?	—	—	—	*	*	*	*	*
INC	INC приемник	*	—	—	—	*	*	*	*	—
Команды вычитания										
SUB	SUB приемник, источник	*	—	—	—	*	*	*	*	*
SBB	SBB приемник, источник	*	—	—	—	*	*	*	*	*
AAS	AAS	?	—	—	—	?	?	*	?	*
DAS	DAS	?	—	—	—	*	*	*	*	*
DEC	DEC приемник	*	—	—	—	*	*	*	*	—
NEG	NEG приемник	*	—	—	—	*	*	*	*	*
CMP	CMP приемник, источник	*	—	—	—	*	*	*	*	*
Команды умножения										
MUL	MUL источник	*	—	—	—	?	?	?	?	*
IMUL	IMUL источник	*	—	—	—	?	?	?	?	*
AAM	AAM	?	—	—	—	*	*	?	*	?
Команды деления										
DIV	DIV источник	?	—	—	—	?	?	?	?	?
IDIV	IDIV источник	?	—	—	—	?	?	?	?	?
AAD	AAD	?	—	—	—	*	*	?	*	?
Команды расширения знака										
CBW	CBW	—	—	—	—	—	—	—	—	—
CWD	CWD	—	—	—	—	—	—	—	—	—

Примечание. * означает изменение значения флага, — означает сохранение, ? — неопределенное состояние.

ФОРМАТЫ АРИФМЕТИЧЕСКИХ ДАННЫХ

ДВОИЧНЫЕ ЧИСЛА

Двоичные числа могут иметь 8 или 16 битов и могут быть со знаком или без знака. У *числа без знака* все 8 или 16 битов представляют его значение. Следовательно, двоичные числа без знака могут принимать значения от 0 до 255

(8-битовые) или до 65535 (16-битовые). У *числа со знаком* старший бит (7 или 15) указывает его знак, а остальные биты содержат значение числа. Следовательно, числа со знаком могут принимать значения от –128 до 127 (8-битовые) или от –32768 до 32767 (16-битовые).

ДЕСЯТИЧНЫЕ ЧИСЛА

Микропроцессор 8088 хранит десятичные числа в виде последовательностей байтов без знака в упакованном или неупакованном формате. Каждый байт упакованного десятичного числа содержит две цифры в *двоично-десятичном* коде BCD (binary-coded decimal). При этом код старшей цифры числа занимает четыре старших бита байта. Следовательно, один упакованный десятичный байт может содержать значения от 00 до 99.

Каждый байт неупакованного десятичного числа содержит только один двоично-десятичный код цифры в четырех младших битах. Следовательно, один неупакованный десятичный байт может содержать лишь значение от 0 до 9. При умножении и делении четыре старших бита должны быть нулевыми, а при сложении или вычитании их значение несущественно.

Как же микропроцессор 8088 узнает, с *каким* видом данных он имеет дело? Пусть требуется сложить два байта. Как он определяет, какие числа они представляют (двоичные числа со знаком, двоичные числа без знака, упакованные десятичные числа или неупакованные десятичные числа)? На самом деле микропроцессор 8088 об этом совершенно не заботится и трактует *все* операнды только как двоичные числа.

Это хорошо в том случае, когда Ваши операнды и в самом деле являются двоичными числами, но если они оказались десятичными, то результаты, конечно, будут ошибочными. Для компенсации таких ошибок микропроцессор 8088 имеет группу команд коррекции, которые обеспечивают получение правильного результата после выполнения операций над десятичными числами. Эти команды будут обсуждаться далее.

ХРАНЕНИЕ ЧИСЕЛ В ПАМЯТИ

Как уже упоминалось, микропроцессор 8088 хранит 16-битовые числа в порядке, противоположном естественному представлению, а именно он хранит младшие биты числа в байте с меньшим адресом. Например, при запоминании числа 1234H в ячейке по имени NUM он размещает 34H по адресу NUM, а 12H – по адресу NUM+1. При чтении изображения (или дампа) содержимого памяти учитывайте эту схему свертки байтов. Запомните фразу: "младший байт – младший адрес, старший байт – старший адрес".

КОМАНДЫ СЛОЖЕНИЯ

КОМАНДА СЛОЖЕНИЯ ADD И КОМАНДА СЛОЖЕНИЯ С ДОБАВЛЕНИЕМ ПЕРЕНОСА ADC

Команды ADD (add – сложить) и ADC (add with carry – сложить с переносом) могут складывать как 8-, так и 16-битовые операнды. Команда ADD складывает содержимое операнда-источника и операнда-приемника и помещает результат в операнд-приемник. В символической нотации ее действия можно

описать как

```
приемник = приемник+источник
```

Команда ADC делает то же, что и команда ADD, но при сложении использует также флаг переноса CF, что можно записать следующим образом:

```
приемник = приемник+источник+перенос
```

Перенос при сложении двоичных чисел аналогичен переносу при сложении десятичных чисел в столбик. Например, при сложении

```
   98
 + 13
   79
 ----
  190
```

возникает два переноса: сложение единиц вызывает добавление 2 к десяткам, а сложение десятков и перенос из столбца единиц вызывает другой перенос, а именно числа 1 в столбец сотен. Перенос возникает тогда, когда сумма цифр столбца в нем не помещается.

Аналогичным образом возникает перенос, когда ЭВМ складывает двоичные числа: если сумма не помещается в операнде-приемнике, то генерируется перенос. Как известно, 8-битовый регистр может содержать значения без знака в диапазоне от 0 до 255. Если мы, например, выполним двоичное сложение чисел 250 и 10, то получим

```
   1111 1010  (двоичное представление числа 250)
 + 0000 1010  (двоичное представление числа 10)
 -----------
 1 0000 01000 (ответ: десятичное значение 260)
```

Результат верен, но занимает 9 двоичных битов! Если при выполнении этой операции мы использовали 8-битовые регистры, то младшие 8 битов будут занесены в регистр-приемник, а девятый бит – во *флаг переноса* CF.

Теперь Вам нетрудно понять, почему микропроцессор 8088 имеет две разные команды сложения. Одна из них (ADD) может складывать значения, представляемые байтами или словами, а также младшие части значений повышенной точности. Другая команда (ADC) используется для сложения старших частей значений повышенной точности.

Например, команда

```
ADD AX,CX
```

складывает 16-битовые значения регистров AX и CX и возвращает результат в регистр AX. Если Ваши операнды имеют длину более 16 битов, то можно воспользоваться последовательностью команд вида

```
ADD AX,CX       ;Сначала сложить младшие 16 битов, а затем
ADC BX,DX       ; старшие 16 битов
```

которая складывает 32-битовое число, находящееся в регистрах CX и DX, с 32-битовым числом, находящимся в регистрах AX и BX. Использованная здесь команда ADC добавляет к (DX)+(BX) любой перенос от сложения (CX)+(AX)[1].

---

[1] Через (AX), (BX), (CX) и (DX) обозначено содержимое соответствующих регистров. – *Прим. перев.*

Вы можете также добавлять находящийся в памяти операнд к регистру и наоборот или добавлять непосредственный операнд к регистру или операнду, находящемуся в памяти. Приведем несколько примеров:

```
ADD AX,MEM_WORD        ;Добавить значение ячейки памяти к регистру
ADD MEM_WORD,AX        ; или наоборот
ADD AL,10              ;Добавить константу к регистру
ADD MEM_BYTE,OFH       ; или к ячейке памяти
```

Допускается большинство возможных комбинаций, но нельзя добавить значение одной ячейки памяти к другой или использовать в качестве приемника непосредственное значение.

Команды ADD и ADC могут воздействовать на шесть флагов:

Флаг переноса CF равен 1, если результат сложения не помещается в операнде-приемнике; в противном случае он равен 0.

Флаг четности PF равен 1, если результат имеет четное число битов со значением 1; в противном случае он равен 0.

Вспомогательный флаг переноса AF равен 1, если результат сложения десятичных чисел требует коррекции; в противном случае он равен 0.

Флаг нуля ZF равен 1, если результат равен 0; в противном случае он равен 0.

Флаг знака SF равен 1, если результат отрицателен (старший бит равен 1); в противном случае он равен 0.

Флаг переполнения OF равен 1, если сложение двух чисел одного знака (оба положительные или оба отрицательные) приводит к результату, который превышает диапазон допустимых значений приемника в обратном коде, а сам приемник при этом меняет знак. В противном случае флаг OF равен 0.

Флаги SF и OF имеют смысл только при сложении чисел со знаком, а флаг AF – только при сложении десятичных чисел.

Микропроцессор 8088 имеет команды, которые проверяют флаги и на основе результатов проверки принимают решение о том, куда передать управление. Например, при отрицательном результате (SF=1) должна исполняться одна группа команд, а при положительном (SF=0) – другая. Эти команды "принятия решения" будут обсуждаться ниже.

КОРРЕКЦИЯ РЕЗУЛЬТАТА СЛОЖЕНИЯ ДЛЯ ПРЕДСТАВЛЕНИЯ В КОДАХ ASCII И В УПАКОВАННОМ ДЕСЯТИЧНОМ ФОРМАТЕ (КОМАНДЫ AAA И DAA)

Как уже упоминалось, при выполнении сложения микропроцессор 8088 рассматривает операнды как двоичные числа. Что же произойдет, если они будут двоично-десятичными кодами чисел (кратко десятичными или BCD-числами)? Разберемся в этом на примере. При сложении упакованных BCD-чисел 26 и 55 микропроцессор 8088 выполнит следующее двоичное сложение:

```
  0010 0110  (BCD-число 26)
+ 0101 0101  (BCD-число 55)
-----------
  0111 1011  (??)
```

Вместо правильного значения (BCD-число 81) мы получим результат, у которого старшая цифра 7, а младшая – шестнадцатеричная цифра B. Означает ли это, что нельзя складывать десятичные числа? Нет, это означает лишь то, что результат должен быть *скорректирован* для представления в десятичной форме.

Коррекция результата сложения десятичных чисел осуществляется командами AAA (ASCII adjust for addition – скорректировать результат сложения для представления в кодах ASCII) и DAA (Decimal adjust for addition – скорректировать сложение для представления в десятичной форме). В них не требуется наличия операнда: предполагается, что корректируемое значение находится в регистре AL.

Команда AAA преобразует содержимое регистра AL в правильную *неупакованную* десятичную цифру в младших четырех битах регистра AL (и заполняет нулями старшие четыре бита). Она используется в следующем контексте:

```
ADD AL,BL        ;Сложить неупакованные числа, находящиеся в AL и BL
AAA              ; и преобразовать результат в неупакованное число
```

Если результат превышает 9, то команда AAA добавляет 1 к содержимому регистра AH (чтобы учесть избыточную цифру) и полагает флаг CF равным 1; в противном случае она обнуляет флаг CF. Кроме того, команда AAA изменяет состояние флага AF и оставляет значения флагов PF, ZF, SF и OF неопределенными. Но так как в данном случае только флаг CF имеет смысл, то считайте значения остальных флагов уничтоженными.

Команда DAA преобразует содержимое регистра AL в две правильные *упакованные* десятичные цифры. Она используется в следующем контексте:

```
ADD AL,BL        ;Сложить упакованные BCD-числа в AL и BL
DAA              ; и преобразовать результат в упакованное число
```

Если результат превышает предельное значение для упакованных BCD-чисел (99), то команда DAA добавляет 1 к содержимому регистра AH и полагает флаг CF равным 1. Кроме того, команда DAA изменяет состояния флагов PF, AF, ZF и CF и оставляет значение флага OF неопределенным. Но так как в данном случае только флаг CF имеет смысл, то считайте остальные пять флагов уничтоженными.

КОМАНДА ПРИРАЩЕНИЯ ЗНАЧЕНИЯ ПРИЕМНИКА НА ЕДИНИЦУ (INC)

Команда INC (increment – прирастить) добавляет 1 к содержимому регистра или ячейки памяти, но в отличие от команды ADD не воздействует на флаг переноса CF. Команда INC удобна для приращения значений счетчиков в циклах команд. Ее можно использовать и для приращения значения индексного регистра или указателя при доступе к последовательно расположенным ячейкам памяти

Приведем несколько примеров:

```
INC CX               ;Прирастить значение 16-битового
INC AL               ; или 8-битового регистра
INC MEM_BYTE         ;Прирастить значение байта
INC MEM_WORD[BX]     ; или слова памяти
```

Как ни странно, приращение значения 8-битового регистра отнимает у микропроцессора 8088 больше времени, чем приращение значения 16-битового регистра (см. характеристики команды INC в приложении B)! Это вызвано тем, что разработчики фирмы Intel предполагали, что программисты будут чаще пользоваться счетчиками размером в слово, а не байт, и предусмотрели специальную однобайтовую версию команды INC для 16-битовых регистров.

ВЫПОЛНЕНИЕ ВЫЧИТАНИЯ МИКРОПРОЦЕССОРОМ 8088

Внутри микропроцессора 8088, как и любого другого микропроцессора общего назначения, нет устройства вычитания. Однако он имеет устройство сложения (сумматор) и может вычитать числа путем сложения. Хотя это и может показаться странным, тем не менее это концепция, как сказал бы Шерлок Холмс, "элементарна".

Чтобы понять, как можно вычитать путем сложения, посмотрим, как вычесть 7 из 10. В начальной школе учат записывать это как

10–7,

но в старших классах (скажем, в курсе алгебры) учат и другому способу записи:

10+(–7).

Первым способом (непосредственное вычитание) вычитание может быть выполнено микропроцессором, имеющим устройство вычитания. Так как микропроцессор 8088 его не имеет, то он вычитает в два приема. Сначала он меняет знак у вычитаемого (у второго числа), т.е. *обращает* его, а затем складывает уменьшаемое и обращенное вычитаемое. Так как микропроцессор 8088 оперирует двоичными числами, то обращение знака числа производится путем так называемого *дополнения до двух.*

Чтобы выполнить дополнение до двух, берется исходная форма двоичного числа и значение каждого его бита обращается (каждый 0 заменяется на 1, а 1 – на 0), а затем к полученному числу добавляется 1.

Применяя это к нашему примеру, получаем 8-битовые представления чисел 10 и 7: 00001010B и 00000111B соответственно. Затем дополним двоичное представление 7 до двух:

```
  1111 1000  (обратить все биты)
+         1  (добавить 1)
  ---------
  1111 1001  (дополнение до двух числа 7,или –7).
```

Теперь операция вычитания примет следующий вид:

```
  0000 1010  (10)
+ 1111 1001  (–7)
  ---------
  0000 0011  (Ответ: 3)
```

Эврика! Мы получили правильный ответ!

Так как микропроцессор 8088 выполняет дополнение до двух автоматически, то Вам эта операция понадобится в редких случаях. Позже в этом разделе мы рассмотрим команду NEG, посредством которой можно выполнить дополнение до двух, если оно когда-либо Вам понадобится.

КОМАНДЫ ВЫЧИТАНИЯ SUB И ВЫЧИТАНИЯ С ЗАЕМОМ SBB

Команды SUB (substract – вычесть) и SBB (substract with borrow – вычесть с заемом) аналогичны соответственно командам сложения ADD и ADC, только при вычитании флаг переноса CF действует как признак *заема*. Команда SUB вычитает операнд-источник из операнда-приемника и возвращает результат в операнд-приемник, т.е.

```
приемник = приемник–источник
```

Команда SBB делает то же самое, но дополнительно вычитает значение флага переноса CF:

```
приемник = приемник-источник-перенос .
```

Как и в случае сложения, команды вычитания выполняют две отдельные функции. Первая команда SUB вычитает числа размером в байт или слово, а также младшие биты чисел повышенной точности. Другая команда SBB вычитает старшие биты чисел повышенной точности.

Например, команда

```
SUB AX,CX
```

вычитает содержимое регистра CX из содержимого регистра AX и возвращает результат в регистр AX.

Если размеры операндов превышают 16 битов, то пользуйтесь последовательностью команд вида

```
SUB AX,BX     ;Вычесть младшие 16 битов,
SBB BX,DX     ; а затем - старшие 16 битов
```

Здесь мы вычитаем 32-битовое число, помещенное в регистры CX и DX, из 32-битового числа, помещенного в регистры AX и BX. При вычитании содержимого регистра DX из содержимого регистра BX команда SBB учитывает возможность заема при выполнении первого вычитания.

Можно вычитать из содержимого регистра содержимое ячейки памяти (и наоборот) или вычитать из содержимого регистра либо ячейки памяти непосредственное значение. Ниже приведены примеры допустимых команд:

```
SUB AX,MEM_WORD        ;Вычесть из регистра содержимое ячейки памяти
SUB MEM_WORD[BX],AX    ; или наоборот
SUB AL,10              ;Вычесть константу из регистра
SUB MEM_BYTE,OFH       ; или из ячейки памяти
```

Нельзя непосредственно вычесть значение одной ячейки из другой или использовать непосредственное значение как приемник.

Команды SUB и SBB могут воздействовать на шесть флагов следующим образом:

Флаг переноса CF равен 1, если требуется заем; в противном случае он равен 0.

Флаг четности PF равен 1, если результат вычитания имеет четное число битов со значением 1; в противном случае он равен 0.

Вспомогательный флаг переноса AF равен 1, если результат вычитания десятичных чисел требует коррекции; в противном случае он равен 0.

Флаг нуля ZF равен 1, если результат равен 0; в противном случае он равен 0.

Флаг знака SF равен 1, если результат отрицателен (старший бит равен 1); в противном случае он равен 0.

Флаг переполнения OF равен 1, если при вычитании чисел, имеющих разные знаки, результат превышает диапазон значений приемника в обратном коде, а сам приемник изменяет знак; в противном случае флаг OF равен 0.

Флаги SF и OF имеют смысл только при вычитании чисел со знаком, а флаг AF – только при вычитании десятичных чисел.

КОРРЕКЦИЯ РЕЗУЛЬТАТА ВЫЧИТАНИЯ ДЛЯ ПРЕДСТАВЛЕНИЯ В КОДАХ ASCII И В УПАКОВАННОМ ДЕСЯТИЧНОМ ФОРМАТЕ (КОМАНДЫ AAS И DAS)

При вычитании, как и при сложении, микропроцессор 8088 рассматривает операнды как двоичные числа. Поэтому вычитание чисел, представленных в двоично-десятичном коде (BCD-чисел), может привести к неправильным результатам. Предположим, например, что надо вычесть BCD-число 26 из BCD-числа 55. Микропроцессор 8088 выполнит двоичное вычитание следующим образом: дополнит до двух двоично-десятичное представление числа 26, а затем выполнит сложение:

```
  0101 0101  (BCD-число 55)
+ 1101 1010  (дополнение до двух BCD-числа 26)
 ----------
 1 0010 1111  (? ?).
```

Вместо правильного значения (BCD-числа 29) мы получили результат, у которого старшая цифра 2, младшая цифра – шестнадцатеричная цифра F, и при этом бит переноса равен 1. Конечно, этот результат требует коррекции.

Коррекция результата вычитания двух десятичных чисел осуществляется командами AAS (ASCII adjust for substraction – скорректировать вычитание для представления в кодах ASCII) и DAS (Decimal adjust for substraction – скорректировать вычитание для представления в десятичной форме). При их исполнении предполагается, что корректируемое число находится в регистре AL.

Команда AAS преобразует содержимое регистра AL в правильную *неупакованную* десятичную цифру в младших четырех битах регистра AL (и обнуляет старшие четыре бита). Она используется в следующем контексте:

```
SUB AL,BL      ;Вычесть BCD-число (содержимое BL) из AL
AAS            ; и преобразовать результат в неупакованное число
```

Если результат превышает 9, то команда AAS вычитает 1 из содержимого регистра AH и полагает флаг CF равным 1, в противном случае она обнуляет флаг CF. Кроме того, команда AAS изменяет состояние флага AF и оставляет значения флагов PF, ZF, SF и OF неопределенными. Но так как в данном случае только флаг CF имеет смысл, то считайте значения остальных флагов уничтоженными.

Команда DAS преобразует содержимое регистра AL в две правильные упакованные десятичные цифры. Она используется в следующем контексте:

```
SUB AL,BL      ;Вычесть упакованное BCD-число (содержимое) BL из AL
DAS            ; и преобразовать результат в упакованное число
```

Если результат превышает предельное значение для упакованных BCD-чисел (99), то команда DAS вычитает 1 из содержимого регистра AH и полагает флаг CF равным 1; в противном случае она обнуляет флаг CF. Кроме того, команда DAS изменяет состояния флагов PF, AF, ZF и SF, а значение флага OF оставляет неопределенным. Но так как в данном случае только флаг CF имеет смысл, то считайте остальные упомянутые флаги уничтоженными.

КОМАНДА УМЕНЬШЕНИЯ СОДЕРЖИМОГО ПРИЕМНИКА НА ЕДИНИЦУ DEC

Команда DEC (decrement – уменьшить) вычитает 1 из содержимого регистра или ячейки памяти, но при этом (в отличие от команды SUB) не воздействует на флаг переноса CF. Команда DEC часто используется в циклах для уменьшения значения счетчика до тех пор, пока оно не станет нулевым или отрицательным. Ее можно использовать также для уменьшения значения индексного регист-

ра или указателя при доступе к последовательно расположенным ячейкам памяти.

Приведем несколько примеров:

```
DEC CX              ;Уменьшить значение 16-битового
DEC AL              ; или 8-битового регистра
DEC MEM_BYTE        ;Уменьшить значение байта
DEC MEM_WORD[BX]    ; или слова памяти
```

КОМАНДА ОБРАЩЕНИЯ ЗНАКА NEG

Команда NEG вычитает значение операнда-приемника из нулевого значения и тем самым формирует его дополнение до двух. Команда NEG оказывает на флаг то же действие, что и команда SUB. Но поскольку один из операндов равен 0, то можно точнее описать условия изменения состояний флагов. Итак, при исполнении команды NEG флаги изменяются следующим образом:

Флаг переноса CF и флаг знака SF равны 1, если операнд представляет собой ненулевое положительное число; в противном случае они равны 0.

Флаг четности PF равен 1, если результат имеет четное число битов, равных 1; в противном случае он равен 0.

Флаг нуля ZF равен 1, если операнд равен 0; в противном случае он равен 0.

Флаг переполнения OF равен 1, если операнд-байт имеет значение 80H или операнд-слово имеет значение 8000H; в противном случае он равен 0.

Команда NEG полезна для вычитания значения регистра или ячейки памяти из непосредственного значения. Например, Вам нужно вычесть значение регистра AL из 100. Так как непосредственное значение не может служить приемником, то команда SUB 100, AL недопустима. В качестве альтернативы можно обратить знак содержимого регистра AL и добавить к нему 100:

```
NEG AL
ADD AL,100
```

КОМАНДА СРАВНЕНИЯ ЗНАЧЕНИЙ ИСТОЧНИКА И ПРИЕМНИКА CMP

Большая часть программ выполняет команды вовсе не в том порядке, в котором они хранятся в памяти. Обычно в программах есть переходы, циклы вызовы процедур и другие команды, заставляющие микропроцессор 8088 передавать управление из одного места памяти в другое. Команды, вызывающие такие передачи, будут рассмотрены ниже.

А сейчас мы обсудим команду CMP (compare – сравнить), которая обычно используется для изменения состояния флагов, на основании которых команды передачи управления "принимают решение" передавать или не передавать управление.

Подобно команде SUB команда CMP вычитает операнд-источник из операнда-приемника и в зависимости от результата устанавливает или обнуляет флаги (табл. 3.5). Но в отличие от команды SUB *команда CMP не сохраняет результат вычитания*.

Другими словами, команда CMP не изменяет операнды. Она целиком предназначена для установки значений флагов, на основании которых команды условного перехода будут "принимать решение" о передаче управления.

Таблица 3.5. Результаты исполнения команды CMP

Условие	OF	SF	ZF	CF
Операнды без знака				
Источник < приемник	Н	Н	0	0
источник = приемник	Н	Н	1	0
источник > приемник	Н	Н	0	1
Операнды со знаком				
Источник < приемник	0/1	0	0	Н
Источник = приемник	0	0	1	Н
Источник > приемник	0/1	1	0	Н

Примечание. Н означает "не имеет значения", 0/1 означает, что флаг может быть равен 0 или 1 в зависимости от значений операндов.

КОМАНДЫ УМНОЖЕНИЯ

КОМАНДА УМНОЖЕНИЯ ЧИСЕЛ БЕЗ ЗНАКА MUL И ЦЕЛОГО УМНОЖЕНИЯ ЧИСЕЛ СО ЗНАКОМ IMUL

Если Вам когда-либо приходилось терпеть муки составления программы умножения для микропроцессоров Z80, 6502 или для каких-либо других распространенных 8-битовых микропроцессоров, то Вы будете рады узнать, что микропроцессор 8088 имеет встроенные команды умножения. Команда MUL (multiply – умножить) умножает числа без знака, а IMUL (integer multiply – умножить целые числа) – числа со знаком. Обе команды могут умножать как байты, так и слова.

Эти команды имеют формат

```
MUL источник
IMUL источник
```

где **источник** – регистр общего назначения или ячейка памяти размером в байт или слово. В качестве второго операнда команды MUL и IMUL используют содержимое регистра AL (при операциях над байтами) или регистра AX (при операциях над словами). Произведение имеет двойной размер и возвращается следующим образом:

Умножение *байтов* возвращает 16-битовое произведение в регистрах AH (старший байт) и AL (младший байт).

Умножение *слов* возвращает 32-битовое произведение в регистрах DX (старшее слово) и AX (младшее слово).

По завершении исполнения этих команд флаги переноса CF и переполнения OF показывают, какая часть произведения существенна для последующих операций. После исполнения команды MUL флаги CF и OF равны 0, если старшая половина произведения равна 0; в противном случае оба этих флага равны 1. После исполнения команды IMUL флаги CF и OF равны 0, если старшая половина произведения

представляет собой лишь расширение знака младшей половины. В противном случае они равны 1.

Приведем несколько примеров умножения:

```
MUL BX           ;Умножить BX на AX без знака
MUL MEM_BYTE     ;Умножить содержимое ячейки памяти на AL без знака
IMUL DL          ;Умножить DL на AL со знаком
IMUL MEM_WORD    ;Умножить содержимое ячейки памяти на AX со знаком
```

Команды MUL и IMUL не позволяют в качестве операнда использовать непосредственное значение. Такое значение перед умножением надо загрузить в регистр или в ячейку памяти. Например, в результате исполнения команд

```
MOV DX,10
MUL DX
```

содержимое регистра AX будет умножено на 10.

КОРРЕКЦИЯ РЕЗУЛЬТАТОВ УМНОЖЕНИЯ ДЛЯ ПРЕДСТАВЛЕНИЯ В КОДАХ ASCII (КОМАНДА AAM)

Команда AAM (ASCII adjust for multiplication – скорректировать умножение для представления в кодах ASCII) преобразует результат предшествующего умножения байтов в два правильных неупакованных десятичных операнда. Она считает, что произведение двойного размера находится в регистрах AH и AL, и возващает неупакованные операнды в регистрах AH и AL. Чтобы команда AAM работала правильно, исходные множимое и множитель должны быть правильными неупакованными байтами.

Для выполнения преобразования команда AAM делит значение регистра AL на 10 и запоминает частное и остаток в регистрах AH и AL соответственно. Кроме того, она модифицирует флаг четности PF, флаг нуля ZF и флаг знака SF в зависимости от полученного значения регистра AL. Состояние флага переноса CF, вспомогательного флага AF и флага переполнения становятся неопределенными.

Рассмотрим действие команды AAM на примере. Пусть регистр AL содержит 9 (0000 1001B), а регистр BL – 7 (0000 0111B). Команда

```
MUL BL
```

умножит значение регистра AL на значение регистра BL и возвратит 16-битовый результат в регистрах AH и AL. В нашем случае она возвратит 0 в регистре AH и 00111111B (десятичное 63) в регистре AL.

Следующая за ней команда

```
AAM
```

поделит значение регистра AL на 10 и возвратит частное 0000 0110B в регистре AH, а остаток 0000 0011B в регистре AL. Тем самым мы получаем правильный результат: BCD-число 63 в неупакованном формате.

У микропроцессора 8088 нет команды умножения *упакованных* десятичных чисел. Для выполнения этой операции сначала распакуйте эти числа, перемножьте их и воспользуйтесь командой AAM, а затем упакуйте результат.

КОМАНДА ДЕЛЕНИЯ ЧИСЕЛ БЕЗ ЗНАКА DIV И ДЕЛЕНИЯ ЧИСЕЛ СО ЗНАКОМ IDIV

Имея две отдельные команды умножения, микропроцессор 8088 имеет и две отдельные команды деления. Команда DIV (divide – разделить) выполняет деление чисел без знака, а команда IDIV (integer divide – разделить целые числа) выполняет деление чисел со знаком.

Эти команды имеют формат

```
DIV источник
IDIV источник
```

где **источник** – делитель размером в байт или слово, находящийся в регистре общего назначения или в ячейке памяти. Делимое должно иметь двойной размер; оно извлекается из регистров AH и AL (при делении на 8-битовое число) или из регистров DX и AX (при делении на 16-битовое число). Результаты возвращаются следующим образом:

Если операнд-источник представляет собой *байт*, то частное возвращается в регистре AL, а остаток в регистре AH.

Если операнд-источник представляет собой *слово*, то частное возвращается в регистре AX, а остаток – в регистре DX.

Обе команды оставляют состояние флагов неопределенными, но если частное не помещается в регистре-приемнике (AL или AX), то микропроцессор 8088 сообщает Вам об этом весьма драматическим образом: он генерирует *прерывание типа 0 (деление на 0)*.

Переполнение результата деления возникает при следующих условиях:

1. Делитель равен 0.
2. При делении байтов без знака делимое по меньшей мере в 256 раз превышает делитель.
3. При делении слов без знака делимое по меньшей мере в 65 536 раз превышает делитель.
4. При делении байтов со знаком частное лежит вне диапазона от –128 до +127.
5. При делении слов со знаком частное лежит вне диапазона от –32768 до 32767.

Приведем несколько типичных примеров операций деления:

```
DIV BX          ;Разделить DX:AX на BX, без знака
DIV MEM_BYTE    ;Разделить AH:AL на байт памяти, без знака
IDIV DL         ;Разделить AH:AL на DL со знаком
IDIV MEM_WORD   ;Разделить DX:AX на слово памяти, со знаком
```

Команды DIV и IDIV не позволяют прямо разделить на непосредственное значение; его надо предварительно загрузить в регистр или ячейку памяти. Например, команды

```
MOV BX,20
DIV BX
```

разделят объединенное содержимое регистров DX и AX на 20.

КОМАНДА КОРРЕКЦИИ ДЕЛЕНИЯ ДЛЯ ПРЕДСТАВЛЕНИЯ В КОДАХ ASCII (КОМАНДА AAD)

Все ранее описанные команды десятичной коррекции (AAA, DAA, AAS, DAS и AAM) выполняли действия над результатом операции. В противоположность им команда AAD (ASCII adjust for division – скорректировать деление для

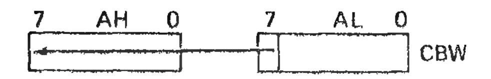




Рис. 3.7. Расширение знака командами CBW и CWD

представления в кодах ASCII) должна исполняться непосредственно перед операцией деления.

Команда ADD преобразует неупакованное делимое в двоичное значение и загружает его в регистр AL. Для этого она умножает старшую цифру делимого (содержимое регистра AH) на 10 и добавляет полученный результат к младшей цифре, находящейся в регистре AL. Затем она обнуляет содержимое регистра AH.

Приведем типичный пример применения команды AAD:

```
AAD             ;Скорректировать неупакованное делимое в AH:AL,
DIV BL          ; а затем выполнить деление
```

КОМАНДЫ РАСШИРЕНИЯ ЗНАКА

Существуют две команды, позволяющие выполнять операции над смешанными данными за счет удвоения размера операнда со знаком. Команда CBW (convert byte to word – преобразовать байт в слово) воспроизводит 7-й бит регистра AL во всех битах регистра AH; команда CWD (convert word to doubleword – преобразовать слово в двойное слово) воспроизводит 15-й бит регистра AX во всех битах регистра DX. Эти действия иллюстрирует рис. 3.7.

Таким образом, команда CBW позволяет сложить байт и слово, вычесть слово из байта и т.д. Аналогично команда CWD позволяет разделить слово на слово. Приведем несколько примеров:

```
CBW             ;Сложить байт в AL со словом в BX
ADD AX, BX

CBW             ;Умножить байт в AL на слово в BX
IMUL BX

CWD             ;Разделить слово в AX на слово в BX
IDIV BX
```

3.6. КОМАНДЫ МАНИПУЛИРОВАНИЯ БИТАМИ

Эти команды манипулируют группами битов в регистрах или ячейках памяти. В табл. 3.6 они разделены на три группы: логические команды, команды сдвига и команды циклического сдвига.

ЛОГИЧЕСКИЕ КОМАНДЫ

Эти команды названы логическими потому, что они действуют по правилам формальной логики, а не арифметики. Например, логическое утверждение "если A истинно и B истинно, то C истинно" находит свое отражение в команде

Таблица 3.6. Команды манипулирования битами

Мнемокод	Формат	Флаги								
		OF	DF	IF	TF	SF	ZF	AF	PF	CF
Логические команды										
AND	AND приемник, источник	0	—	—	—	*	*	?	*	0
OR	OR приемник, источник	0	—	—	—	*	*	?	*	0
XOR	XOR приемник, источник	0	—	—	—	*	*	?	*	0
NOT	NOT приемник	—	—	—	—	—	—	—	—	—
TEST	TEST приемник, источник	0	—	—	—	*	*	?	*	0
Команды сдвига										
SAL/SHL	SAL приемник, счетчик	*	—	—	—	*	*	?	*	*
SAR	SAR приемник, счетчик	*	—	—	—	*	*	?	*	*
SHR	SHR приемник, счетчик	*	—	—	—	0	*	?	*	*
Команды циклического сдвига										
ROL	ROL приемник, счетчик	*	—	—	—	—	—	—	—	*
ROR	ROR приемник, счетчик	*	—	—	—	—	—	—	—	*
RCL	RCL приемник, счетчик	*	—	—	—	—	—	—	—	*
RCR	RCR приемник, счетчик	*	—	—	—	—	—	—	—	*

Примечание. * означает изменение значения флага, — означает сохранение, ? — неопределенное состояние.

AND (И) микропроцессора 8088, которая применяет его к соответствующим битам двух операндов.

Точнее говоря, команда AND полагает равным 1 все те биты операнда-приемника, в позициях которых содержится 1 у обоих операндов. А те биты приемника, в

Таблица 3.7. Шестнадцатеричные значения битов

Номер бита	Шестнадцатеричное значение	Номер бита	Шестнадцатеричное значение
0	0001	8	0100
1	0002	9	0200
2	0004	10	0400
3	0008	11	0800
4	0010	12	1000
5	0020	13	2000
6	0040	14	4000
7	0080	15	8000

позициях которых содержится любая другая комбинация значений (или 00, или 0 и 1), полагаются равными нулю.

Так как логические операции манипулируют битами операндов, то обычно при записи значений таких операндов используют шестнадцатеричную систему счисления. Логические команды микропроцессора 8088 могут оперировать байтами или словами, поэтому обычно приходится иметь дело с двузначными или четырехзначными шестнадцатеричными числами.

Чтобы Вам было легче конструировать правильные значения *масок* для логических операций, в табл. 3.7 показаны шестнадцатеричные представления значения 1 в каждом из 16 битов слова. Например, при операциях над битом 2 правильное значение маски равно 4H; при операциях над битами 2 и 3 значение маски равно 0CH (шестнадцатеричная сумма 4+8) и т.д.

ЛОГИЧЕСКИЕ КОМАНДЫ AND, OR И XOR

Мнемокоды этих команд должны показаться Вам знакомыми, поскольку в разд. 2.7 мы уже обсуждали одноименные логические операции AND (И), OR (ИЛИ) и XOR (Исключающее ИЛИ). Однако эти операции выполняются в процессе трансляции программы, а команды действуют при ее исполнении. Ради полноты изложения мы опишем здесь и команды AND, OR и XOR, а за разъяснением их действия Вы можете обратиться к разд. 2.7.

Операндами команд AND, OR и XOR могут быть байты или слова. В этих командах можно сочетать два регистра, регистр с ячейкой памяти или непосредственное значение с регистром или ячейкой памяти. Результаты выполнения этих команд приведены в табл. 3.8.

Команда AND маскирует (обнуляет) некоторые биты, после чего можно выполнить дальнейшую обработку остальных битов. Как уже упоминалось, в каждой позиции бита, где оба операнда содержат 1, операнд-приемник также будет содержать 1. В тех же позициях, где операнды имеют любую другую комбинацию значений, операнд-приемник будет содержать 0. Запомните, что *при исполнении команды AND биты операнда-приемника становятся равны 0 всюду, где операнд-источник содержит 0, и сохраняются там, где операнд-источник содержит 1.*

Приведем несколько примеров команд AND:

```
AND AX,BX              ;Выполнить AND над двумя регистрами
AND AL,MEM_BYTE        ;Выполнить AND над регистром и ячейкой памяти
AND MEM_BYTE           ; или наоборот
AND BL,1101B           ;Выполнить AND над константой и регистром
AND TABLE[BX],MASK3    ; или ячейкой памяти
```

Таблица 3.8. **Результаты исполнения команд AND, OR и XOR**

Источник	Приемник	Результат		
		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Рассмотрим следующий пример применения команды AND. Пусть порт 200 соединен с 16-битовым регистром состояния внешнего устройства системы и бит 6 показывает, включено (1) или выключено (0) устройство. Если Ваша программа может продолжать работу только в случае, когда устройство включено, то она должна содержать следующий цикл:

```
CHK_PWR:  IN   AX,200       ;Прочитать состояние устройства
          AND  AX,1000000B  ;Изолировать индикатор включения
          JZ   CHK_PWR      ;Подождать включения питания,
          ...               ; а затем продолжить работу
          ...
```

Команда JZ (jump if zero – перейти если нуль), которая нами еще не рассматривалась, заставляет микропроцессор 8088 вернуться к команде IN с меткой CHK_PWR, если флаг нуля ZF равен 1, и перейти к следующей команде в противном случае. В нашем примере флаг ZF равен 1 только в том случае, когда индикатор включения (бит 6) равен 1, так как команда AND обнуляет все остальные биты регистра AX.

Команда OR полагает равным 1 те биты операнда-приемника, в позициях которых хотя бы один из операндов содержит 1. Эта команда обычно используется для принудительного присваивания 1 заданным битам. Например, команда

```
OR BX,OC00H
```

полагает два старших бита (14 и 15) регистра BX равными 1 и оставляет другие биты не измененными.

Команда XOR используется, если надо выяснить, в каких битах значения операндов различаются, или если надо обратить состояния заданных битов. Команда XOR полагает равным 1 все те биты приемника, в позициях которых операнды имеют различные значения, иначе говоря, те биты, в позициях которых один из операндов имеет значение 0, а другой – 1. Если оба операнда содержат в данной позиции либо 0, либо 1, то команда XOR обнуляет этот бит приемника.

Например, команда

```
XOR BX,OC000H
```

обращает состояния двух старших битов регистра BX (14 и 15) и сохраняет остальные биты неизмененными.

КОМАНДА ЛОГИЧЕСКОГО ОТРИЦАНИЯ NOT

Команда NOT (НЕ) обращает состояние каждого бита регистра или ячейки памяти и ни на какие флаги не воздействует. Таким образом, команда NOT заменяет каждый 0 на 1, а каждую 1 – на 0. Другими словами, она выполняет для операнда *дополнение до единицы*.

КОМАНДА ПРОВЕРКИ TEST

Команда TEST (test – проверить) выполняет операцию AND над операндами, но воздействует только на флаги и не изменяет значения операндов. Команда TEST изменяет флаги точно так же, как команда AND: она обнуляет флаги CF и OF, изменяет флаги PF, ZF и SF, а флаг AF оставляет неопределенным.

Когда вслед за командой TEST указана команда JNZ (jump if not zero – перейти, если не нуль), переход произойдет только в том случае, если хотя бы в одной позиции бита оба операнда содержат 1.

КОМАНДЫ СДВИГА И ЦИКЛИЧЕСКОГО СДВИГА

У микропроцессора 8088 есть семь команд, осуществляющих сдвиг 8- или 16-битового содержимого регистров или ячеек памяти на одну или несколько позиций влево или вправо. Три из них *сдвигают* операнд, а остальные четыре его *вращают* или *циклически сдвигают*.

Для всех семи команд флаг переноса CF является как бы расширением операнда битом 9 или битом 17. Иначе говоря, флаг CF приобретает значение бита, сдвинутого за один из концов операнда. Команды сдвига и циклического сдвига вправо помещают во флаг CF значение нулевого бита. Команды сдвига и циклического сдвига влево помещают в него значение бита 7 (при операциях над байтом) или бита 15 (при операциях над словом).

Команды сдвига и циклического сдвига распадаются на две группы. *Логические* команды сдвигают операнд, не считаясь с его знаком; они используются для действий над числами без знака или над нечисловыми значениями, например над масками. *Арифметические* команды сохраняют старший, знаковый бит операнда; они используются для действий над числами со знаком. На рис. 3.8 показано действие этих команд.

Команды сдвига и циклического сдвига имеют два операнда: *приемник* и *счетчик*. Приемником может быть 8- или 16-битовый регистр общего назначения или ячейка памяти. Счетчик может быть цифрой 1 или значением без знака в регистре CL.

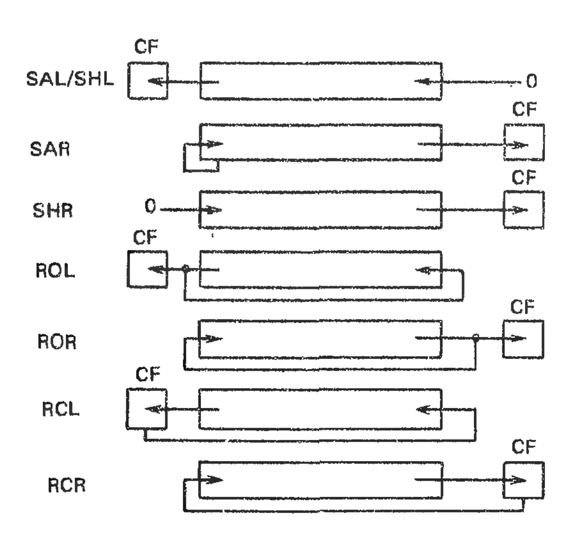


Рис. 3.8. Команды сдвига и циклического сдвига

Команды SAL (shift aritmetic left – сдвинуть влево арифметически) и SAR (shift aritmetic right – сдвинуть вправо арифметически) сдвигают числа со знаком. Команда SAR сохраняет знак операнда, репродуцируя его при выполнении сдвига. Команда SAL не сохраняет знак, но заносит 1 во флаг переполнения OF в случае изменения знака операнда. При каждом сдвиге операнда команда SAL заносит 0 в вакантный нулевой бит этого операнда.

Команды SHL (shift logical left – сдвинуть влево логически) и SHR (shift logical right – сдвинуть вправо логически) сдвигают числа без знака. Команда SHL идентична команде SAL. Команда SHR аналогична команде SHL, но сдвигает операнд не влево, а вправо. При каждом сдвиге операнда команда SHR заносит 0 в вакантный старший бит этого операнда (бит 7 при сдвиге байта, бит 15 при сдвиге слова).

Помимо флагов CF и OF команды сдвига изменяют флаги PF, ZF и SF, а флаг AF оставляют неопределенным.

Чтобы познакомиться с работой команд сдвига, предположим, что регистр AL содержит 0B4H, а флаг переноса CF равен 1. В двоичном коде

```
AL = 10110100   CF = 1
```

Команды сдвига воздействуют на регистр AL и флаг CF следующим образом:

```
После SAL AL,1:   AL = 01101000   CF = 1
После SAR AL,1:   AL = 11011010   CF = 0
После SHL AL,1:   AL = 01101000   CF = 1
После SHR AL,1:   AL = 01011010   CF = 0
```

Имеется несколько интересных приложений команд сдвига. Например, в следующем фрагменте команд команда SHL используется для преобразования двух неупакованных BCD-чисел (старшая цифра извлекается из регистра BL, младшая – из регистра AL) в упакованное BCD-число в регистре AL:

```
MOV CL,4      ;Загрузить счетчик сдвига в CL
SHL BL,CL     ;Сдвинуть старшую цифру в старшие четыре бита BL
OR  AL,BL     ;Получить упакованное BCD-число слиянием AL и BL
```

Поскольку *сдвиг операнда на один бит влево удваивает значение операнда (умножает на 2), а сдвиг на один бит вправо уменьшает значение операнда вдвое (делит на 2),* то команды сдвига можно использовать в качестве команд быстрого умножения и деления.

Следующие команды сдвига показывают, каким образом можно разделить на четыре содержимое регистра AX. Во всех случаях предполагается, что регистр CL содержит 2.

```
SHL AX,CL      ;Умножить число без знака на 4
SAL AX,CL      ;Умножить число со знаком на 4
SHR AX,CL      ;Разделить число без знака на 4
SAR AX,CL      ;Разделить число со знаком на 4
```

Применяя команды сдвига вместо команд умножения и деления, можно сэкономить немало времени. Каждая из предыдущих команд сдвига выполняется за 16 тактов. Еще 4 такта требуется для загрузки значения в регистр CL, итого – 20 тактов. Сравнивая это время с минимальными временами исполнения команд MUL (118 тактов), IMUL (128 тактов), DIV (144 такта) и IDIV (165 тактов), мы видим,

что команды сдвига выполняют эти действия в *шесть-восемь раз быстрее*, чем команды умножения и деления!

В то время как отдельная команда сдвига может умножить или разделить только на степень числа 2, манипулирование несколькими регистрами позволяет выполнить умножение или деление на другие числа. Например, приведенная ниже последовательность команд делит содержимое регистра AX на 10:

```
MOV BX,10      ;Сохранить содержимое AX в BX
SHL AX,1       ;Сдвинуть AX (умножить на 2)
SHL AX,1       ;Сдвинуть AX еще раз (умножить на 4)
ADD AX,BX      ;Сложить с исходным значением AX (умножить на 5)
SHL AX,1       ;Сдвинуть AX еще раз (умножить на 10)
```

Хотя в этой последовательности пять команд, но выполняется она в 11 раз быстрее одной команды MUL!

КОМАНДЫ ЦИКЛИЧЕСКОГО СДВИГА

Команды циклического сдвига похожи на команды сдвига, но в отличие от последних *сохраняют* сдвинутые за пределы операнда биты, помещая их обратно в операнд. Как и при исполнении команд сдвига, сдвинутый за пределы операнда бит запоминается во флаге переноса CF.

При исполнении команды ROL (rotate left – сдвинуть влево циклически) и ROR (rotate right – сдвинуть вправо циклически) вышедший за пределы операнда бит входит в него с противоположного конца. При исполнении команд RCL (rotate left through carry – сдвинуть влево циклически вместе с флагом переноса) и RCR (rotate right through carry – сдвинуть вправо циклически вместе с флагом переноса) в противоположный конец операнда помещается значение флага переноса CF. Все команды циклического сдвига воздействуют только на флаги CF и OF.

Чтобы познакомиться с работой команд циклического сдвига, вернемся к исходным данным предыдущего примера:

```
AL = 10110100   CF = 1
```

Команды циклического сдвига воздействуют на регистр AL и флаг CF следующим образом:

```
После ROL AL,1:   AL = 01101001   CF = 1
После ROR AL,1:   AL = 01011010   CF = 0
После RCL AL,1:   AL = 01101001   CF = 1
После RCR AL,1:   AL = 11011010   CF = 0
```

3.7. КОМАНДЫ ПЕРЕДАЧИ УПРАВЛЕНИЯ

Как уже упоминалось, команды хранятся в последовательно расположенных ячейках памяти, но редко исполняются по порядку. Все программы, за исключением простейших, содержат переходы и вызовы процедур, заставляющие микропроцессор изменять путь исполнения программы.

Команды передачи управления обеспечивают переход из одной части программы в другую. Как показано в табл. 3.9, эти команды можно подразделить на три группы: команды безусловной передачи управления, команды условной передачи управления и команды управления циклами.

Таблица 3.9. **Команды передачи управления**

Мнемокод	Формат	Флаги								
		OF	DF	IF	TF	SF	ZF	AF	PF	CF
Команды безусловной передачи управления										
CALL	CALL **имя**	–	–	–	–	–	–	–	–	–
RET	RET [число удаляемых из стека значений]	–	–	–	–	–	–	–	–	–
JMP	JMP **имя**	–	–	–	–	–	–	–	–	–
Команды условной передачи управления										
JA/JNBE	JA **близкая_метка**	–	–	–	–	–	–	–	–	–
JAE/JNB	JAE **близкая_метка**	–	–	–	–	–	–	–	–	–
JB/JNAE/JC	JB **близкая_метка**	–	–	–	–	–	–	–	–	–
JBE/JNA	JBE **близкая_метка**	–	–	–	–	–	–	–	–	–
JCXZ	JCXZ **близкая_метка**	–	–	–	–	–	–	–	–	–
JE/JZ	JE **близкая_метка**	–	–	–	–	–	–	–	–	–
JG/JNLE	JG **близкая_метка**	–	–	–	–	–	–	–	–	–
JGE/JNL	JGE **близкая_метка**	–	–	–	–	–	–	–	–	–
JL/JNGE	JL **близкая_метка**	–	–	–	–	–	–	–	–	–
JLE/JNG	JLE **близкая_метка**	–	–	–	–	–	–	–	–	–
JNC	JNC **близкая_метка**	–	–	–	–	–	–	–	–	–
JNE/JNZ	JNE **близкая_метка**	–	–	–	–	–	–	–	–	–
JNO	JNO **близкая_метка**	–	–	–	–	–	–	–	–	–
JNP/JPO	JNP **близкая_метка**	–	–	–	–	–	–	–	–	–
JNS	JNS **близкая_метка**	–	–	–	–	–	–	–	–	–
JO	JO **близкая_метка**	–	–	–	–	–	–	–	–	–
JP/JPE	JP **близкая_метка**	–	–	–	–	–	–	–	–	–
JS	JS **близкая_метка**	–	–	–	–	–	–	–	–	–
Команды управления циклами										
LOOP	LOOP **близкая_метка**	–	–	–	–	–	–	–	–	–
LOOPE/LOOPZ	LOOPE **близкая_метка**	–	–	–	–	–	–	–	–	–
LOOPNE/LOOPNZ	LOOPNE **близкая_метка**	–	–	–	–	–	–	–	–	–

Примечание. – означает сохранение значения флага.

ПРОЦЕДУРЫ

До сих пор приводимые в данной книге примеры содержали команды, выполнявшиеся только *один* раз. Поэтому можно предположить, что для выполнения специфической операции в разных местах программы в каждом из них надо продублировать последовательность команд, исполняющих эту операцию. Конечно, такое дублирование раздражает и отнимает много времени. Кроме того, оно значительно удлиняет программу.

На самом деле Вы *можете* избежать дублирования, если определите повторяющуюся последовательность команд как *процедуру*. Процедура (или, как часто говорят, подпрограмма) представляет собой совокупность команд, которая написана один раз, но может быть исполнена по мере необходимости в любом месте программы.

Процесс передачи управления из основной части программы в процедуру называется *вызовом*, т.е. процедура вызывается. При вызове процедуры микропроцессор 8088 исполняет ее команды, а затем возвращается к тому месту, откуда был сделан вызов.

Возникают два вопроса: как вызвать процедуру и как микропроцесор 8088 возвращает управление в нужное место программы? Для ответа на них рассмотрим команды

КОМАНДА ВЫЗОВА ПРОЦЕДУРЫ CALL И ВОЗВРАТА ИЗ ПРОЦЕДУРЫ RET

Команды, обеспечивающие исполнение процедур, должны выполнять три функции:

1. Обеспечить сохранение содержимого указателя команд IP. Когда процедура исполнена, находившийся в этом указателе адрес используется микропроцессором 8088 для возврата к месту вызова. Мы будем называть его *адресом возврата*.
2. Заставить микропроцессор начать исполнение процедуры.
3. Использовать сохраненное содержимое указателя команд IP для возврата в программу и обеспечить продолжение ее исполнения с этого места.

Все эти функции выполняются двумя командами: CALL (call a procedure – вызвать процедуру) и RET (return from procedure – возвратиться из процедуры). Они, в сущности, являются эквивалентами на языке ассемблера операторов Бейсика GOSUB и RETURN.

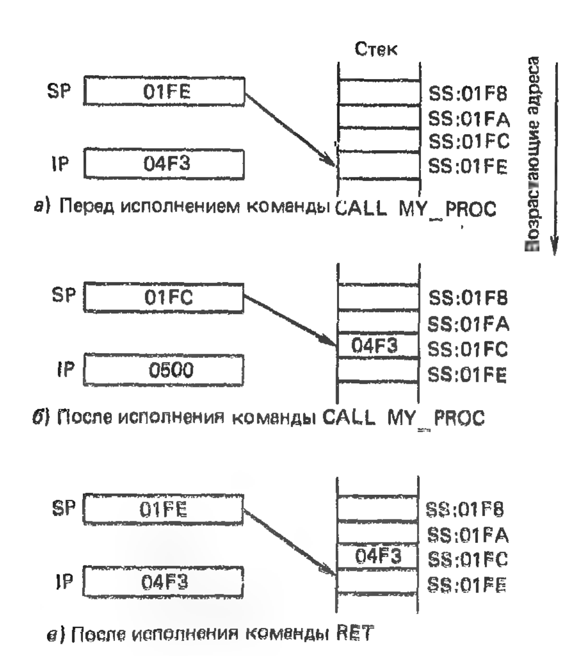
Команда CALL осуществляет функции запоминания адреса возврата и передачи управления процедуре. Она помещает в стек адрес возврата, занимающий 16 битов, если процедура определена с атрибутом NEAR, и 32 бита, если она определена с атрибутом FAR (см. разд. 2.4). Процедуры с атрибутом NEAR могут быть вызваны только из того сегмента, в котором они находятся; процедуры с атрибутом FAR могут быть вызваны и из другого сегмента.

Команда CALL имеет формат

```
CALL имя
```

где **имя** – имя вызываемой процедуры (т.е. метка ее начала). Если процедура **имя** имеет атрибут NEAR, то команда CALL помещает смещение адреса следующей

Рис. 3.9. Воздействие процедуры на стек



команды в стек. Если процедура имя имеет атрибут FAR, то команда CALL помещает в стек содержимое регистра CS, а затем смещение адреса.

После сохранения адреса возврата команда CALL загружает смещение адреса метки имя в указатель команд IP. Если процедура имеет атрибут FAR, то команда CALL загружает также номер блока метки имя в регистр CS.

Команда RET заставляет микропроцессор 8088 возвратиться из процедуры в программу, вызвавшую эту процедуру делая это "откатом" всего, что сделала команда CALL. Команда RET обязательно должна быть последней командой процедуры, исполняемой микропроцессором 8088. (Это не значит, что команда RET должна стоять в конце процедуры – она лишь *исполняется* последней.)

Команда RET извлекает из стека адрес возврата. Если процедура имеет атрибут NEAR (т.е. находится в том же сегменте команд, что и команда CALL), то команда RET извлекает из стека одно слово и загружает его в указатель команд IP. Если процедура имеет атрибут FAR (т.е. находится в другом сегменте команд), то команда RET извлекает из стека два слова: сначала смещение адреса для загрузки в указатель команд IP, а затем номер блока для загрузки в регистр CS.

Например, для вызова процедуры MY_PROC с атрибутом NEAR из некоторого места Вашей программы надо указать следующую последовательность команд (в распечатке указаны также смещения адресов команд):

```
04F0              CALL  MY_PROC   ;Вызвать процедуру
04F3  NEXT        MOV   AX,BX     ;Вернуться сюда из процедуры
                  ..
                  ..
0500  MY_PROC     PROC            (начало процедуры)
0500              MOV   CL,6      ;Первая команда процедуры
                  ..
                  ..
051E              RET             ;Вернуться в вызвавшую программу
                  ..
051F  MY_PROC     ENDP            (конец процедуры)
```

При исполнении команды CALL микропроцессор 8088 помещает в стек смещение адреса метки NEXT (04F3H), затем загружает смещение адреса процедуры MY_PROC (0500H) в указатель команд IP. Так как в псевдооператоре PROC атрибут дистанции не указан, то процедура MY_PROC по умолчанию имеет атрибут NEAR.

Так как содержимое регистра IP изменилось, то микропроцессор 8088 продолжит исполнение с команды, имеющей это новое смещение адреса. В нашем примере такой командой будет

```
MOV CL,6
```

Когда микропроцессор обнаруживает команду RET, то он извлекает адрес возврата из стека и помещает его в указатель команд IP. Это заставляет его возобновить исполнение с команды, имеющей метку NEXT. На рис. 3.9 показаны стек, указатель стека SP и указатель команд IP до и после исполнения команды CALL, а также после исполнения команды RET.

КОСВЕННЫЕ ВЫЗОВЫ ПРОЦЕДУРЫ

До сих пор мы обсуждали только один вид команды CALL, а именно *прямой* вызов, при котором операндом служит метка начала процедуры. Но можно осуществить и *косвенный* вызов процедуры через регистр или ячейку памяти. При косвенных вызовах через ячейку памяти микропроцессор 8088 извлекает значение указателя команд IP для процедуры из сегмента данных, если только Вы не используете регистр BP или не укажете замену сегмента. Если для адресации ячейки памяти Вы используете регистр BP, то микропроцессор 8088 извлечет значение указателя команд IP из сегмента стека.

Вы можете вызвать процедуру с атрибутом NEAR через регистр, например:

```
CALL BX
```

В данном случае регистр BX содержит смещение адреса процедуры относительно регистра сегмента CS. При исполнении этой команды микропроцессор 8088 копирует содержимое регистра BX в указатель команд IP, затем передает управление команде, адресуемой парой регистров CS:IP. Если, например, регистр BX содержит 1ABH, то микропроцессор 8088 извлечет следующую команду из ячейки 1ABH, находящейся в сегменте команд.

Процедуру с атрибутом NEAR можно вызвать косвенно, используя переменную размером в слово, например:

```
CALL WORD PTR [BX]
CALL WORD PTR [BX][SI]
CALL WORD PTR VARIABLE_NAME
CALL WORD PTR VARIABLE_NAME[BX]
CALL MEM_WORD
CALL WORD PTR ES:[BX][SI]
```

Последняя команда CALL получает адрес процедуры из ячейки дополнительного сегмента (благодаря указанию ES:); остальные команды извлекают адреса процедур из ячеек сегмента данных.

Процедуру с атрибутом FAR можно вызвать косвенно, используя переменную размером в двойное слово, например:

```
CALL DWORD PTR[BX]
CALL MEM_DWORD
CALL DWORD PTR SS:VARIABLE_NAME[SI]
```

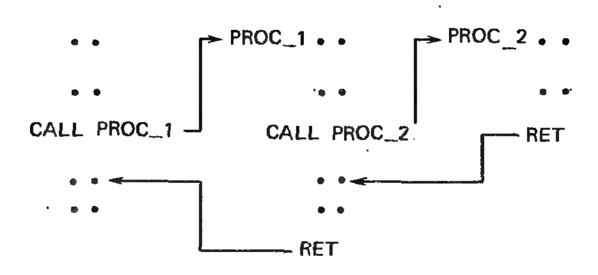


Рис. 3.10. Вложенные процедуры

Здесь первые две команды CALL извлекают адреса процедур из сегмента данных, а последняя – из сегмента стека.

ВЛОЖЕННЫЕ ПРОЦЕДУРЫ

Процедура может сама вызывать другие процедуры. Например, процедура чтения символа с клавиатуры может его декодировать и в зависимости от результата вызвать одну из других процедур. Вызов одной процедуры из другой называется *вложением* процедур. На рис. 3.10 показаны передачи управления в программе, где процедура PROC_1 вызывает процедуру PROC_2 (т.е. процедура PROC_2 вложена в процедуру PROC_1).

Обычно программисты описывают вложение в терминах уровней. О программе, изображенной на рис. 3.10 (или программе такого же типа), где вложение распространяется только на вызов CALL PROC_2 (процедура PROC_2 не вызывает других процедур), говорят, что она имеет один уровень вложения. Однако процедура PROC_2 вполне может вызывать процедуру PROC_3 (два уровня вложения), а процедура PROC_3 может вызывать процедуру PROC_4 (три уровня вложения) и т.д.

Так как каждая команда CALL помещает в стек два или четыре байта адреса, то число уровней вложения ограничено только размером сегмента стека.

Поскольку сегмент стека может иметь до 64 Кбайт, то возможности вложения практически не ограничены.

КОМАНДА БЕЗУСЛОВНОГО ПЕРЕХОДА JMP

Команда JMP (jump unconditionally – перейти безусловно) представляет собой эквивалент на языке ассемблера оператора Бейсика GOTO; она заставляет микропроцессор 8088 извлечь новую команду не из следующей ячейки памяти, а из какой-то другой. Команда JMP имеет формат

```
JMP имя
```

где операнд **имя** подчиняется тем же правилам, что и операнд команды CALL. Иначе говоря, он может иметь атрибут NEAR или FAR, быть прямым или косвенным. При прямом переходе команда JMP занимает три бита, если метка **имя** имеет атрибут NEAR, и пять байтов, если она имеет атрибут FAR. Например, команда

```
JMP THERE
```

занимает три байта, если метка THERE находится в том же сегменте, или пять байтов, если она находится в другом сегменте. (В последнем случае перед сегментом команды JMP должен быть указан псевдооператор EXTRN THERE: FAR, а перед сегментом, содержащим метку, – псевдооператор PUBLIC THERE.)

Если адрес метки находится не далее –128 или +128 байтов от адреса команды JMP, то можно сделать команду JMP *двухбайтовой*, указав, что ее операнд имеет тип SHORT (short – короткий). Например, команда

```
JMP SHORT NEAR_LABEL
```

займет два байта. Она исполняется за то же время, что и команда

```
JMP NEAR_LABEL
```

но занимает в памяти на один байт меньше.

Обычно команда JMP используется для обхода группы команд, которым передается управление из другой части программы. Например, Вам может встретиться применение команды JMP вида

```
         ..
         ..
         MOV   AX,BX
         ADD   DX,AX
         JMP   THERE
HERE     MOV   MEM_WORD,DX
         ..
         ..
THERE    MOV   SAVE_DX,DX
         ..
         ..
```

КОМАНДЫ УСЛОВНОЙ ПЕРЕДАЧИ УПРАВЛЕНИЯ

У микропроцессора 8088 есть 17 различных команд, которые позволяют ему "принять решение" о ходе исполнения программы в зависимости от определенных условий, например нулевого значения регистра или единичного значения флага переноса CF. Если такое условие выполнено, то микропроцессор 8088 выполнит переход; в противном случае он продолжит исполнение со следующей команды программы. Как показано в табл. 3.9, некоторые из этих команд имеют два или три различных мнемокода. (Если учитывать эти альтернативные мнемокоды, то можно было бы заявить, что микропроцессор 8088 имеет 31 команду условной передачи управления. Если Вы считаете это более правильным, то всерьез подумайте о карьере менеджера по сбыту!)

Например, для Ассемблера команды JA LABEL и JNBE LABEL идентичны. Действие этих команд зависит от результата исполнения предшествующей команды сравнения CMP или команд вычитания (SUB или SBB).

Первый мнемокод JA (jump above – перейти при превышении) сообщает микропроцессору 8088, что переход надо выполнять в том случае, если приемник "выше" источника. А второй мнемокод JNBE (jump if not be low nor equal – перейти, если не ниже и не равен) сообщает ему, что переход надо выполнить, если приемник "не ниже" источника и "не равен" ему. Таким образом, команды JA и JNBE сообщают микропроцессору одно и то же, но в разных терминах. Это сделано исключительно для того, чтобы Вы могли написать удобные для чтения и более понятные программы. Команды условной передачи управления имеют следующий

общий формат:

```
Jx  близкая_метка
```

где x – модификатор, состоящий из одной, двух или трех букв. Запись операнда **близкая_метка** подчеркивает, что метка перехода должна находиться не далее –128 или +127 байтов от команды условной передачи управления. Сравните эти команды с командой безусловного перехода JMP, которая может передать управление в любое место памяти.

В табл. 3.10 описаны все команды условной передачи управления и указаны условия, выполнение которых вызывает переход (чтобы облегчить Вам поиск, все мнемокоды указанные отдельно).

Таблица 3.10. **Команды условной передачи управления**

Команда	Описание	Условие перехода
JA	Jump if Above – перейти, если выше	CF=0 и ZF=0
JAE	Jump if Above or Equal – перейти, если выше или равно	CF=0
JB	Jump if Below – перейти, если ниже	CF=1
JBE	Jump if Below or Equal – перейти, если ниже или равно	CF=1 или ZF=1
JC	Jump if Carry – перейти, если перенос	CF=1
JCXZ	Jump if CX is Zero – перейти, если значение регистра CX равно 0	CX=0
JE	Jump if Equal – перейти, если равно	ZF=1
*JG	Jump if Greater – перейти если больше	ZF=0 и SF=OF
*JGE	Jump if Greater or Equal – перейти, если больше или равно	SF=OF
*JL	Jump if Less – перейти, если меньше	SF≠OF
*JLE	Jump if Less or Equal – перейти, если меньше или равно	ZF=1 или SF≠OF
JNA	Jump if Not Above – перейти, если не выше	CF=1 или ZF=1
JNAE	Jump if Not Above nor Equal – перейти, если не выше и не равно	CF=1
JNB	Jump if Not Below – перейти, если не ниже	CF=0
JNBE	Jump if Not Below nor Equal – перейти, если не ниже и не равно	CF=0 и ZF=0
JNC	Jump if No Carry – перейти, если нет переноса	CF=0
JNE	Jump if Not Equal – перейти, если не равно	ZF=0
*JNG	Jump if Not Greater – перейти, если не больше	ZF=1 или SF≠OF
*JNGE	Jump if Not Greater nor Equal – перейти, если не больше и не равно	SF≠OF
*JNL	Jump if Not Less – перейти, если не меньше	SF=OF
*JNLE	Jump if Not Less nor Equal – перейти, если не меньше и не равно	ZF=0 и SF=OF
*JNO	Jump if No Overflow – перейти, если нет переполнения	OF=0
JNP	Jump if No Parity – перейти, если нет четности (нечетная сумма разрядов)	PF=0
*JNS	Jump if No Sign – перейти, если знаковый разряд нулевой	SF=0
JNZ	Jump if Not Zero – перейти, если не нуль	ZF=0
*JO	Jump if Overflow – перейти, если переполнение	OF=1
JP	Jump if Parity – перейти, если есть четность (четная сумма битов)	PF=1
JPE	Jump if Parity Even – перейти, если сумма битов четная	PF=1
JPO	Jump if Parity Odd – перейти, если сумма битов нечетная	PF=0
JS	Jump on Sign – перейти, если знаковый бит равен 1	SF=1
JZ	Jump if Zero – перейти, если нуль	ZF=1

Примечание. Знаком * отмечены команды, относящиеся к действиям над числами со знаком (в обратном коде).

Команды условной передачи управления занимают в памяти два байта: первый байт содержит код операции, а второй – относительный сдвиг. Исполнение этих команд занимает 16 тактов, если происходит переход, и четыре такта, если перехода нет. Поэтому при составлении программы старайтесь подбирать такие команды условной передачи управления, при которых переход менее вероятен.

Приведем несколько примеров команд условной пеоедачи управления

1. Последовательность команд

```
ADD AL,BL
JC  TOOBIG
```

осуществляет переход к метке TOOBIG, если при сложении возник перенос.

2. Последовательность команд

```
SUB AL,BL
JZ  ZERO
```

осуществляет переход к метке ZERO, если при вычитании в регистре AL оказался нулевой результат.

3. Последовательность команд

```
CMP AL,BL
JE  ZERO
```

осуществляет переход к метке ZERO, если значения регистров AL и BL одинаковы. (Здесь можно было бы использовать эквивалентный мнемокод – JZ, но мнемокод JE (jump if equal – перейти, если равно) в данном случае более содержателен.)

4. В некоторых ситуациях приходится выбирать между двумя различными командами условного перехода в зависимости от того, проверяется результат операции над числами без знака или над числами со знаком. Предположим, что Вам требуется перейти к метке BXMORE, если содержимое регистра BX имеет большее значение, чем содержимое регистра AX. Тогда надо использовать последовательность команд

```
CMP BX,AX
JA  BXMORE
```

если операнды *не имеют знака*, и последовательность команд

```
CMP BX,AX
JG  BXMORE
```

если они *имеют знак*.

СОВМЕСТНОЕ ПРИМЕНЕНИЕ КОМАНД УСЛОВНОЙ ПЕРЕДАЧИ УПРАВЛЕНИЯ И КОМАНДЫ СРАВНЕНИЯ CMP

Командам условной передачи управления могут предшествовать любые команды, изменяющие состояния флагов, но обычно они используются совместно с командой сравнения CMP. В табл. 3.5 (разд. 3.5) показано, как команда CMP воздействует на флаги при разных соотношениях между источником и приемником. Теперь, после описания широкого спектра команд условной передачи управления, более "практичной" будет табл. 3.11, в которой показано, какими условными переходами надо пользоваться при возможных сочетаниях значений источника и приемника.

Ниже (в примере 3.1) для иллюстрации типичного приложения комбинации сравнение/условная передача показан фрагмент программы, размещающий в памяти два числа без знака в порядке возрастания; предполагается, что смещения

Таблица 3.11. **Применение команд условной передачи управления в сочетании с командой CMP**

Условие перехода	Следующая за CMP команда	
	для чисел без знака	для чисел со знаком
Приемник больше источника	JA	JG
Приемник равен источнику	JE	JE
Приемник не равен источнику	JNE	JNE
Приемник меньше источника	JB	JL
Приемник меньше источника или равен ему	JBE	JLE
Приемник больше источника или равен ему	JAE	JGE

адресов этих чисел в сегменте данных находятся в регистрах BX и DI соответственно. Напоминаем, что для перестановки двух чисел в памяти одно из них необходимо загрузить в регистр, поскольку микропроцессор 8088 не имеет команды пересылки типа *память-память.*

Одна команда сравнения может взаимодействовать с двумя командами условной передачи управления для выделения трех альтернатив: "меньше", "равно" и "больше". Пример 3.2 показывает фрагмент программы, в котором одна из трех групп команд выбирается в зависимости от того, будет значение регистра AL меньше, равно или больше 10.

В этом фрагменте с помощью команды JAE проверяется условие "AL выше или равно 10". Если оно выполнено, то микропроцессор 8088 переходит к метке AE10. Затем с помощью команды JA определяется, лежит ли значение регистра AL "выше" 10. Если это так, то микропроцессор 8088 переходит к метке A10. Обычно первые две группы команд завершаются командой JMP, обеспечивающей обход команд остальных групп.

Пример 3.1. Размещение двух чисел в порядке возрастания

```
; Этот фрагмент располагает два 16-битовых числа без знака
; в памяти в порядке возрастания: меньшее число заносится в
; ячейку с меньшим адресом. Смещение адреса первого числа
; берется из регистра BX; смещение авдреса второго числа -
; из регистра DI.

        MOV   AX,[BX]     ;Загрузить первое число в AX
        CMP   AX,[DI]     ;Сравнить его со вторым числом
        JBE   DONE        ;Первое число меньше второго или
                          ;равно ему?
        XCHG  AX,[DI]     ;Если нет, обменяться значениями
        MOV   [BX],AX
DONE:   ...
        ...
```

Пример 3.2. Выбор трех альтернатив

```
; Этот фрагмент выполняет одну из трех различных групп команд
; в зависимости от того, будет ли число без знака, находящееся
; в регистре AL, меньше 10, больше или равно ему

         CMP   AL,10        ;Сравнить AL с 10
         JAE   AE10
         ...                ;Команды для выполнения при AL < 10
         ...
AE10:    JA    A10
         ...                ;Команды для выполнения при AL = 10
         ...
A10:     ...                ;Команды для выполнения при AL > 10
         ...
```

КОМАНДЫ УПРАВЛЕНИЯ ЦИКЛАМИ

Команды управления циклами обеспечивают условные передачи управления при организации циклов. У микропроцессора 8088 регистр счетчика CX служит счетчиком числа повторений циклов. Каждая команда управления циклами уменьшает содержимое регистра CX на 1, а затем использует его новое значение для "принятия решения" о выполнении или не выполнении перехода.

Основная команда этой группы LOOP (loop until Count complete – повторять цикл до конца счетчика) уменьшает содержимое регистра CX на 1 и передает управление операнду **близкая_метка**, если содержимое регистра CX не равно 0. Например, для стократного выполнения определенной группы команд можно воспользоваться следующей конструкцией:

```
         MOV CX,100        ;Загрузить число повторений в CX
START:   ...                 (Повторяемая группа команд)
         ...
         LOOP START        ;Если CX не равен 0, перейти к метке START,
         ...               ; в противном случае выйти из цикла
```

Команда LOOP завершает выполнение цикла только в том случае, если содержимое регистра CX уменьшено до 0. Однако во многих приложениях требуются такие циклы, которые должны завершаться при выполнении определенных условий до того, как содержимое регистра CX достигнет нуля. Такое альтернативное завершение цикла обеспечивается командами LOOPE (loop if equal – повторять цикл, если равно) и LOOPNE (loop if not equal – повторять цикл, если не равно).

Команда LOOPE, имеющая синоним LOOPZ (loop if zero – повторять цикл, если нуль), уменьшает содержимое регистра CX на 1, а затем осуществляет переход, если содержимое регистра CX не равно 0 и флаг нуля ZF равен 1. Таким образом, повторение цикла завершается, если либо содержимое регистра CX равно 0, либо флаг ZF равен 0, либо оба они равны 0. Обычно команда LOOPE используется для поиска первого ненулевого результата в серии операций. Это иллюстрируется примером 3.3, представляющим собой фрагмент программы для поиска первого ненулевого байта в блоке памяти. Смещения адресов первого и последнего байтов блока находятся соответственно в регистрах BX и DI.

Команда LOOPNE, имеющая синоним LOOPNZ (loop if not zero – повторять цикл, пока не нуль), уменьшает содержимое регистра CX на 1, затем осуществляет переход, если содержимое регистра CX не равно 0 и флаг нуля ZF равен 0. Таким образом, повторение цикла завершается, если либо содержимое регистра CX равно

0, либо флаг ZF равен 1, либо будет выполнено и то, и другое. Обычно команда LOOPNE используется для поиска первого нулевого результата всерии операций. Если в примере 3.3 заменить команду LOOPE на команду LOOPNE, то вместо поиска первого ненулевого байта этот фрагмент обеспечит поиск первого нулевого байта блока памяти.

Пример 3.3. Поиск ненулевой ячейки в блоке памяти

```
; Этот фрагмент находит первый ненулевой байт в заданном блоке
; памяти. Смещение начального адреса блока берется из регистра
; BX; смещение конечного адреса блока берется из регистра DI
; Смещение адреса ненулевого байта возвращается в регистре BX
; Если ненулевой байт не найден, то по возвращению BX будет
; содержать то же значение, что и DI
;
           SUB  DI,BX              ;Счетчик байтов =
           INC  DI                 ; (DI) - (BX) + 1
           MOV  CX,DI              ;Занести счетчик байтов в CX
           DEC  BX
NEXT:      INC  BX                 ;Передвинуть указатель к следующей
           CMP  BYTE PTR [BX],0    ; ячейке и сравнить ее с 0
           LOOPE  NEXT             ;Перейти к сравнению следующего байта
           JNZ  NZ_FOUND           ;Найден ненулевой байт?
           ...                     ; Нет. (BX) = (DI)
           ...
NZ_FOUND:  ...                     ; Да. Смещение адреса ненулевого
           ...                     ; элемента находится в BX
```

3.8. КОМАНДЫ ОБРАБОТКИ СТРОК

Команды обработки строк позволяют производить действия над блоками байтов или слов памяти. Эти блоки (или *строки*) могут иметь длину до 64 Кбайт и состоять из числовых значений (двоичных или BCD), алфавитно-цифровых значений (типа символов в кодах ASCII), а также из любых других значений, которые могут храниться в памяти в виде двоичных кодов.

Команды обработки строк предоставляют возможность выполнения пяти основных операций, называемых *примитивами*, которые обрабатывают строку по одному элементу (байту или слову) за прием. Эти примитивы (пересылка, сравнение, сканирование, загрузка и сохранение) описаны в табл. 3.12.

Обратите внимание на то, что каждый примитив представлен тремя разными командами. Первая из них имеет один или два операнда (например, MOVS имеет два операнда), а две остальные не имеют операндов (например, MOVSB и MOVSW). Микропроцессор 8088 может исполнять только те команды обработки строк, которые не имеют операндов. При трансляции программы Ассемблер всегда преобразует команду с операндами в одну из команд без операндов.

Микропроцессор 8088 предполагает, что строка-приемник находится в дополнительном сегменте, а строка-источник – в сегменте данных. Процессор адресует строку-приемник через регистр DI, а строку-источник – через регистр SI. Например, команда MOVSB копирует байт сегмента данных, адрес которого находится в регистре SI, в ячейку дополнительного сегмента, адрес которой находится в регистре DI. По-видимому, фирма Intel выбрала названия DI и SI потому, что они являются легко запоминаемыми аббревиатурами от Destination Index (индекс приемника) и Source Index (индекс источника). Остроумно или нет?

Кстати, невзирая на то, что микропроцессор 8088 ожидает строку-приемник в дополнительном сегменте, а строку-источник – в сегменте данных, вы *можете*

Таблица 3.12. **Команды обработки строк**

Мнемокод	Формат	Флаги								
		OF	DF	IF	TF	SF	ZF	AF	PF	CF
Префиксы повторения										
REP	REP	—	—	—	—	—	—	—	—	—
REPE/REPZ	REPE	—	—	—	—	—	—	—	—	—
REPNE/REPNZ	REPNE	—	—	—	—	—	—	—	—	—
Пересылка										
MOVS	MOVS **строка_приемник, строка_источник**	—	—	—	—	—	—	—	—	—
MOVSB	MOVSB	—	—	—	—	—	—	—	—	—
MOVSW	MOVSW	—	—	—	—	—	—	—	—	—
Сравнение										
CMPS	CMPS **строка_приемник, строка_источник**	*	—	—	—	*	*	*	*	*
CMPSB	CMPSB	*	—	—	—	*	*	*	*	*
CMPSW	CMPSW	*	—	—	—	*	*	*	*	*
Сканирование										
SCAS	SCAS **строка_приемник**	*	—	—	—	*	*	*	*	*
SCASB	SCASB	*	—	—	—	*	*	*	*	*
SCASW	SCASW	*	—	—	—	*	*	*	*	*
Загрузка и сохранение										
LODS	LODS **строка_источник**	—	—	—	—	—	—	—	—	—
LODSB	LODSB	—	—	—	—	—	—	—	—	—
LODSW	LODSW	—	—	—	—	—	—	—	—	—
STOS	STOS **строка_приемник**	—	—	—	—	—	—	—	—	—
STOSB	STOSB	—	—	—	—	—	—	—	—	—
STOSW	STOSW	—	—	—	—	—	—	—	—	—

Примечание. — означает сохранение значения флага, * — его изменение.

использовать и другие комбинации сегментов. Мы расскажем об этом позже.

Так как команды манипулирования строками предназначены для действий над *группой* элементов, то они автоматически модифицируют указатели для адресации следующего элемента строки. Например, команда MOVS увеличивает или уменьшает указатели строки-источника SI и строки-приемника DI после каждого цикла своего исполнения.

Бит флага направления DF в регистре флагов микропроцессора 8088 определяет, будут значения регистров SI и DI увеличены или уменьшены по завершении выполнения команды манипулирования строками. Если флаг DF равен 0, то значения регистров SI и DI *увеличиваются* после исполнения каждой команды; если флаг DF равен 1, то они *уменьшаются*.

Пусть, например, команда MOVS копирует элемент строки-источника в строку-приемник. Если флаг DF равен 0, то микропроцессор 8088 *увеличивает* значения регистров SI и DI после пересылки и тем самым адресуется к следующим элементам памяти. Если флаг DF равен 1, то микропроцессор уменьшает значения регистров SI и DI после пересылки и тем самым адресуется к предыдущему элементу памяти.

Состоянием флага DF можно управлять с помощью двух команд: CLD (clear direction flag – сбросить флаг направления), которая полагает его равным нулю, и STD (установить флаг направления), которая присваивает ему значение 1. Эти команды будут обсуждаться в разделе 3.10. Можно сделать так, чтобы одна команда обработки строк обработала группу последовательных элементов памяти. Для этого перед ней надо указать *префикс повторения* (см. табл. 3.12). Он представляет собой не команду, а однобайтовый модификатор, который заставляет микропроцессор 8088 выполнить аппаратные повторения команды обработки строк, что значительно сокращает время на обработку длинных строк по сравнению с программно-организованными циклами.

ПРЕФИКСЫ ПОВТОРЕНИЯ

Префиксы повторения заставляют микропроцессор 8088 повторять команду обработки строк. Число повторений извлекается из регистра CX. Например, последовательность команд

```
       MOV  CX,500
 REP   MOVS DEST,SOURCE
```

заставит микропроцессор 8088 выполнить команду MOVS 500 раз (эту команду мы обсудим чуть позже), уменьшая значение регистра CX после каждого повторения. В сущности, с помощью префикса REP дается указание повторять, пока не обнаружится конец строки, т.е. повторять, пока значение регистра CX не станет равным нулю.

Остальные префиксы повторения используются при "решении" о продолжении или прекращении повторений флаг нуля ZF. Следовательно, они приложимы только к командам сравнения строк и поиска значения в строке, которые воздействуют на флаг ZF. Префикс REPE (repeat while equal – повторять, пока не равно), имеющий синоним REPZ (repeat while zero – повторять, пока не нуль), повторяет команду, пока флаг ZF равен 1 и значение регистра CX не равно 0. Если приписать префикс REPE *команде сравнения строк* CMPS, то операция сравнения будет повторяться до первого несовпадения. Например, последовательность команд

```
       MOV  CX,100
  REPE CMPS DEST,SOURCE
```

поэлементно сравнивает строки SOURCE и DEST до тех пор, пока не будет просмотрено 100 пар элементов или пока микропроцессор 8088 не найдет в строке DEST элемент, не совпадающий с соответствующим элементом строки SOURCE.

Действия префикса REPNE (repeat while not equal – повторять, пока не равно), имеющего синоним REPNZ (repeat while not zero – повторять, пока не нуль), противоположно действию префикса REPE. Иначе говоря, префикс REPNE обеспечивает повторение модифицированной им команды, пока флаг ZF равен 0 и значение регистра CX не равно 0.

Например, последовательность команд

```
        MOV  CX,100
REPNE CMPS DEST,SOURCE
```

сравнивает строки SOURCE и DEST до тех пор, пока не будет просмотрено 100 пар элементов или пока микропроцессор 8088 не найдет в строке DEST элемент, *совпадающий* с соответствующим элементом в строке SOURCE.

КОМАНДЫ ПЕРЕСЫЛКИ СТРОК

КОМАНДА ПЕРЕСЫЛКИ СТРОКИ MOVS

Команда MOVS копирует байт или слово из одной части памяти в другую. Она имеет формат

```
MOVS строка_приемник,строка_источник
```

Здесь **строка_источник** – строка в сегменте данных, а **строка_приемник** – строка в дополнительном сегменте. Как и в случае команды CMPS, микропроцессор 8088 использует регистр SI для адресации в сегменте данных и регистр DI для адресации в дополнительном сегменте. Таким образом, *команда MOVS копирует байт или слово в дополнительный сегмент*.

Можно заменить сегмент, соответствующий регистру SI, но не регистру DI. Например, можно добиться копирования строки из одной части дополнительного сегмента в другую.

Хотя сама по себе команда MOVS пересылает только один элемент, с помощью префикса REP можно этой командой переслать строку размером до 64 Кбайт (32К слов). Пример 3.4 представляет собой фрагмент программы, копирующий 100-байтовую строку SOURCE из сегмента данных в строку DEST дополнительного сегмента. Как показано в этом примере, каждая групповая пересылка с помощью команды MOVS осуществляется с помощью следующих пяти шагов:

1. Обнулить флаг DF (командой CLD) или установить его (командой STD) в зависимости от того, будет ли пересылка осуществляться от младших адресов к старшим или наоборот.
2. Загрузить смещение адреса строки-источника в регистр SI.
3. Загрузить смещение адреса строки-приемника в регистр DI.
4. Загрузить счетчик элементов (число пересылаемых байтов или слов) в регистр CX.
5. Исполнить команду MOVS с префиксом REP.

Конечно, Ваша программа должна включать в себя *дополнительный* сегмент, вмещающий строку-приемник, и *сегмент данных*, вмещающий строку-источник. Обычно для строки-приемника память резервируется псевдооператором вида

```
DEST DB 100 DUP(?)
```

Как узнает Ассемблер, что надо переслать – байты или слова? Он определяет это по *типу* меток источника и приемника, указанных в поле операнда.

Если эти метки были определены с помощью псевдооператоров DB, то Ассемблер преобразует MOVS в команду MOVSB. Если они были определены с помощью псевдооператоров DW, то Ассемблер преобразует MOVS в команду MOVSW. Таким образом, если Вы определите строки SOURCE и DEST с помощью псевдооператоров DW, то фрагментом программы из примера 3.4 вместо 100 байтов скопируется 100 слов.

Пример 3.4. Групповая пересылка байтов

```
; Этот фрагмент копирует 100 байтов из строки SOURCE, находящуюся
; в сегменте данных, в строку DEST, находящуюся в дополнительном
; сегменте

        CLD                     ;Положить DF = 0 для обработки слева
                                ;направо
        LEA  SI,SOURCE          ;Занести смещение адреса SOURCE в SI,
        LEA  DI,ES:DEST         ; а смещение адреса DEST - в DI
        MOV  CX,100             ;Занести счетчик элементов в CX
REP     MOVS DEST,SOURCE        ;Скопировать байты
```

КОМАНДЫ ПЕРЕСЫЛКИ БАЙТОВ MOVSB И ПЕРЕСЫЛКИ СЛОВ MOVSW

Какое значение имеют операнды команды MOVS кроме того, что они напоминают Вам, какие строки вовлечены в пересылку? Что они сообщают Ассемблеру? Эти операнды лишь дают ему возможность преобразовать команду MOVS в одну из двух форм, непосредственно исполняемых микропроцессором 8088: MOVSB (move byte string – переслать строку байтов) или MOVSW (move word string – переслать строку слов). Для выполнения этого преобразования Ассемблеру надо знать только размер элементов строк, а не их адреса (которые содержатся в регистрах SI и DI).

Так как размер элементов – единственная полезная информация, извлекаемая Ассемблером из команды MOVS, то почему бы не воспользоваться непосредственно командами MOVSB и MOVSW, ориентированными на конкретный размер элементов? В действительности применение команд MOVSB и MOVSW даже предпочтительнее, поскольку в этом случае Ассемблеру нет необходимости выяснить размер операндов. Как можно было ожидать, в этих командах не требуются операнды. Они имеют следующий формат:

```
MOVSB
MOVSW
```

Например, мы могли бы переписать команды из примера 3.4 в следующем виде:

```
      CLD
      LEA SI,SOURCE
      LEA DI,ES:DEST
      MOV CX,100
REP   MOVSB
```

ЗАМЕНА СЕГМЕНТА

Обычно регистр SI адресуется к сегменту данных, но с помощью префикса *замены сегмента* в операнде-источнике можно использовать другой сегмент. Например, команды

```
LEA SI,ES:HERE
LEA DI,ES:THERE
MOVSB
```

копируют байт из строки HERE в строку THERE, где обе строки находятся в дополнительном сегменте.

Так как нельзя заменить сегмент, к которому адресуется регистр DI, то создается впечатление, что приемник всегда должен находиться в дополнительном сегменте. Означает ли это, что нельзя копировать строки в сегменте данных? Нет, строки *можно* копировать и в сегменте данных, но с помощью некоторого трюка. Чтобы скопировать строки в сегменте данных, надо загрузить в регистр дополнительного сегмента ES значение, равное содержимому регистра данных DS. После этого при исполнении команды MOVS микропроцессор 8088 будет считать, что копирует (как обычно) строку из сегмента данных в дополнительный сегмент. Но *Вы-то* знаете, что в действительности он будет копировать строку из одного места сегмента данных в другое!

В примере 3.5 показано, как этот трюк используется для копирования 100 байтов из строки SOURCE_D в строку DEST_D, где обе строки находятся в сегменте данных. Обратите внимание, что за исключением первых двух команд этот фрагмент идентичен примеру 3.4. Этот трюк применим и к другим командам обработки строк, обсуждаемым в настоящем разделе.

Пример 3.5. Пересылка строк в сегменте данных

```
; Этот фрагмент копирует 100 байтов из строки SOURCE_D в строку
; DEST_D, где обе строки находятся в сегменте данных

        PUSH  DS            ;Заставить ES указывать на сегмент данных
        POP   ES
        CLD                 ;Положить DF = 0 для обработки слева
                            ; направо
        LEA   SI,SOURCE_D   ;Занести смещение адреса SOURCE_D в SI,
        LEA   DI,DEST_D     ; а смещение адреса DEST_D - в DI
        MOV   CX,100        ;Занести счетчик элементов в CX
REP     MOVSB               ;Скопировать байты
```

КОМАНДЫ СРАВНЕНИЯ СТРОК

КОМАНДА СРАВНЕНИЯ СТРОК CMPS

Подобно команде сравнения CMP, обсуждавшейся в разд. 3.5, команда сравнения строк CMPS (compare string – сравнить строку) сопоставляет операнд-источник с операндом-приемником и возвращает результат через флаги. Команда CMPS, как и команда CMP, не изменяет значения операндов.

Команда CMPS имеет формат

```
CMPS строка_приемник,строка_источник
```

где **строка_источник** – строка в сегменте данных, адресуемая регистром SI, а **строка_приемник** – строка в дополнительном сегменте, адресуемая регистром DI. Следовательно, команда CMPS сравнивает элемент (байт или слово) сегмента данных с элементом дополнительного сегмента.

Подобно команде CMP команда CMPS сравнивает операнды с помощью их вычитания. Однако CMP вычитает операнд-источник из операнда-приемника, а *CMPS вычитает операнд-приемник из операнда-источника.* Это означает, что

Таблица 3.13. Применение команд условной передачи управления в сочетании с командой CMPS

Условие перехода	Следующая за CMPS команда	
	для чисел без знака	для чисел со знаком
Источник больше приемника	JA	JG
Источник равен приемнику	JE	JE
Источник не равен приемнику	JNE	JNE
Источник меньше приемника	JB	JL
Источник меньше приемника или равен ему	JBE	JLE
Источник больше приемника или равен ему	JAE	JGE

указываемые после команды CMPS команды условной передачи управления должны отличаться от тех, что в аналогичной ситуации следовали бы за командой CMP. Конкретные сведения об этом можно найти в табл. 3.13.

Для сравнения нескольких элементов команду CMPS надо использовать с префиксом повторения. В данном случае префикс REP не имеет смысла, поскольку при его применении во флагах будет возвращен всего лишь результат сравнения двух последних элементов. С командой CMPS надо использовать префиксы REPE (REPZ) или REPNE (REPNZ).

Префикс REPE заставит микропроцессор 8088 сравнивать строки до тех пор, пока либо значение регистра CX не станет равным нулю, либо не будет найдена пара несовпадающих элементов. Например, команды

```
      CLD
      MOV  CX,100
REPE  CMPS DEST,SOURCE
```

будут сравнивать до 100 пар элементов строк SOURCE и DEST, пытаясь найти два несовпадающих элемента.

Префикс REPNE заставит микропроцессор 8088 сравнивать строки до тех пор, пока либо значение регистра CX не станет равным нулю, либо не будет найдена пара совпадающих элементов. Например, команды

```
       CLD
       MOV  CX,100
REPNE  CMPS DEST,SOURCE
```

будут сравнивать до 100 пар элементов строк SOURCE и DEST, пытаясь найти два совпадающих элемента.

Как и в случае команды MOVS, флаг направления DF определяет, обрабатываются строки в порядке возрастания (DF=0) или убывания (DF=1) адресов элементов, а регистры SI и DI соответствующим образом изменяются после каждой операции.

ПРОВЕРКА РЕЗУЛЬТАТОВ СРАВНЕНИЯ

Повторение операций сравнения может завершиться в двух случаях: если значение регистра CX стало равным 0 или флаг ZF стал равен 0 (префикс REPE) либо 1 (префикс REPNE), а Вам может понадобиться узнать, какой из случаев имел место. Выяснить, что же привело к прекращению сравнений, легче всего, указав после команды CMPS команду условной передачи управления, проверяющую значение флага ZF, а именно JE (JZ) или JNE (JNZ).

Например, следующая последовательность команд заставит микропроцессор 8088 перейти к метке NOT_FOUND, если среди первых 100 элементов строк DEST и SOURCE нет ни одной совпадающей пары:

```
            CLD
            MOV    CX,100
REPNE       CMPS   DEST,SOURCE      ;Найти совпадение
            JNE    NOT_FOUND        ;Совпадение обнаружено?
            ...                     ; Да. Продолжить обработку
            ...                     ; здесь
NOT_FOUND:  ...                     ; Нет. Продолжить обработку
            ...                     ; здесь
```

КОМАНДЫ СРАВНЕНИЯ СТРОК БАЙТОВ CMPSB И СТРОК СЛОВ CMPSW

Ассемблер преобразует команду CMPS либо в команду CMPSB (при сравнении байтов), либо в команду CMPSW (при сравнении слов). Рекомендуем Вам вместо команды CMPS использовать эти команды, ориентированные на конкретный размер элементов и не имеющие операндов.

КОМАНДЫ СКАНИРОВАНИЯ СТРОК

Команды сканирования строк позволяют осуществить поиск заданного значения в строке, находящейся в дополнительном сегменте. Смещение адреса первого элемента строки должно быть помещено в регистр DI.

При сканировании строки байтов искомое значение должно находиться в регистре AL, а при сканировании строки слов – в регистре AX. Операция сканирования представляет собой не что иное, как сравнение с содержимым аккумулятора, и воздействует на флаги точно так же, как команды сканирования строк.

КОМАНДА СКАНИРОВАНИЯ СТРОКИ SCAS

Основная команда группы команд сканирования строк SCAS (scan string – сканировать строку) имеет формат

```
SCAS строка_приемник
```

где **строка_приемник** идентифицирует строку в дополнительном сегменте, смещение адреса которой находится в регистре DI. Этот операнд сообщает микропроцессору 8088, что представляет собой искомое значение – байт в регистре AL или же слово в регистре AX.

Как и в случае команды CMPS, для выполнения действий более чем над одним элементом строки надо воспользоваться префиксами повторения REPE (REPZ) или REPNE (REPNZ). Например, с помощью последовательности команд

```
        CLD
        LEA  DI,ES:B_STRING
        MOV  AL,' '
        MOV  CX,100
REPE    SCAS B_STRING
```

можно просмотреть до 100 элементов строки байтов B_STRING в поисках элемента, отличного от пробела. Если такой элемент обнаружен, то смещение адреса следующего за ним элемента возвращается в регистре DI, а флаг нуля ZF полагается равным 0. Последующая команда JNE покажет, найден такой элемент (отсутствие перехода) или не найден (наличие перехода).

КОМАНДЫ СКАНИРОВАНИЯ СТРОКИ БАЙТОВ SCASB И СТРОКИ СЛОВ SCASW

Ассемблер преобразует команду SCAS либо в команду SCASB (при поиске байта), либо в команду SCASW (при поиске слова). Рекомендуем Вам вместо команды SCAS использовать эти команды, ориентированные на конкретный размер элементов и не имеющие операндов.

КОМАНДЫ ЗАГРУЗКИ И СОХРАНЕНИЯ СТРОКИ

После отыскания элемента с помощью команды сравнения или сканирования строки с ним обычно требуется выполнить какие-либо действия. Может понадобиться загрузить этот элемент в регистр (скажем, чтобы точно определить его значение) или изменить его. Эти операции могут быть выполнены с помощью команд загрузки и сохранения строки.

КОМАНДА ЗАГРУЗКИ LODS

Команда LODS (load string – загрузить строку) пересылает операнд строка_источник, адресованный регистром SI, из сегмента данных в регистр AL (при пересылке байта) или в регистр AX (при пересылке слова), а затем изменяет регистр SI так, чтобы он указывал на следующий элемент строки. Его значение увеличивается, если флаг направления DF равен 0, и уменьшается, если флаг DF равен 1.

Например, приведенный ниже фрагмент программы сравнивает строки DEST и SOURCE длиной 500 байтов каждая в поисках первой пары несовпадающих элементов. Если обнаружено несовпадение, то элемент строки SOURCE загружается в регистр AL.

```
        CLD
        LEA     DI,ES:DEST      ;Взять смещение адреса DEST
        LEA     SI,SOURCE       ; и SOURCE
        MOV     CX,500          ;Счетчик элементов
REPE    CMPSB                   ;Искать до несовпадения элементов
        JCXZ    MATCH           ;Несовпадение обнаружено?
        DEC     SI              ; Да. Подправить регистр SI,
        LODS    SOURCE          ; считать элемент в регистр AL
        ...                     ; и обработать его
        ...
MATCH:  ...                     ; Несовпадения нет. Продолжить
        ...                     ; обработку здесь
```

Команда LODS оперировала байтами, следовательно, она либо увеличивала значение регистра SI на 1 (при DF=0), либо уменьшала его на 1 (при DF=1).

Как обычно, команда LODS имеет сокращенные формы LODSB (load byte string – загрузить строку байтов) и LODSW (load word string – загрузить строку слов).

КОМАНДА СОХРАНЕНИЯ СТРОКИ STOS

Команда STOS (Store string – сохранить строку) пересылает байт из регистра AL или слово из регистра AX в элемент операнда строка_приемник, находящийся в дополнительном сегменте и адресуемый регистром DI, а затем изменяет значение регистра DI так, чтобы оно указывало на следующий элемент строки. Это значение увеличивается, если флаг направления DF равен 0, и уменьшается, если флаг DF равен 1.

Будучи повторяемой, команда STOS удобна для заполнения строки заданным значением. Например, следующий фрагмент программы сканирует строку W_STRING длиной в 200 слов в поисках первого ненулевого элемента. Если такой элемент обнаружен, то он и следующие за ним пять слов обнуляются.

```
        CLD
        LEA     DI,ES:W_STRING  ;Адрес строки
        MOV     AX,0            ;Искомое значение = 0
        MOV     CX,200          ;Счетчик поиска = 200 слов
REPNE   SCASW                   ;Сканировать строку
        JCXZ    ALLO            ;Найдено ненулевое слово?
        SUB     DI,2            ; Да. Подправить регистр DI
        MOV     CX,6            ;  и заполнить шесть слов
REP     STOS    W_STRING        ;  нулями
ALLO:   ...                     ; Нет. Продолжить здесь
        ...
```

3.9 КОМАНДЫ ПРЕРЫВАНИЯ

Подобно вызову процедуры прерывание заставляет микропроцессор 8088 сохранить в стеке информацию для последующего возврата, а затем перейти к группе команд, находящейся в некоторой ячейке памяти. Но при вызове процедуры микропроцессор 8088 исполняет процедуру, а при прерывании – *программу обработки прерывания.*

В то время как вызываемая процедура может иметь атрибуты NEAR или FAR, а ее вызов бывает прямым или косвенным, прерывание всегда вызывает косвенный переход к своей программе обработки за счет получения ее адреса из *вектора прерывания* (32-битовой ячейки памяти). Кроме того, вызовы процедуры сохраняют в стеке только адрес, а прерывания сохраняют еще и флаги (так же, как команда PUSHF).

Прерывания могут быть инициированы внешним устройством системы или специальной командой прерывания из программы. У микропроцессора 8088 есть три различные команды прерывания – две команды вызова и одна команда возврата (табл. 3.14).

Таблица 3.14. **Команды прерываний**

Мнемокод	Формат	Флаги								
		OF	DF	IF	TF	SF	ZF	AF	PF	CF
INT	INT тип_прерывания	–	–	0	0	–	–	–	–	–
INTO	INTO	–	–	0	0	–	–	–	–	–
IRET	IRET	*	*	*	*	*	*	*	*	*

Примечание. — означает сохранение значения флага, * — его изменение.

КОМАНДА ПРЕРЫВАНИЯ INT

Команда INT (interrupt – прерывать) имеет формат

```
INT тип_прерывания
```

где **тип_прерывания** – номер, идентифицирующий один из 256 различных векторов, находящихся в памяти. (Таким образом, каждому из 256 прерываний, обсуждавшихся в гл. 1, соответствует один вектор.)

При исполнении команды INT микропроцессор 8088 производит следующие действия:

1. Помещает в стек регистр флагов.
2. Обнуляет флаг трассировки TF и флаг включения-выключения прерываний IF для исключения пошагового режима исполнения команд и блокировки других маскируемых прерываний.
3. Помещает в стек значение регистра CS.
4. Вычисляет адрес вектора прерывания, умножая **тип_прерывания** на 4.
5. Загружает второе слово вектора прерывания в регистр CS.
6. Помещает в стек значение указателя команд IP.
7. Загружает в указатель команд IP первое слово вектора прерывания.

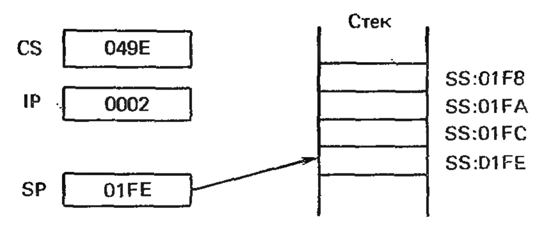
Итак, после исполнения команды INT в стеке окажутся значения регистра флагов и регистров CS и IP, флаги TF и IF будут равны 0, а пара регистров CS:IP будет указывать на начальный адрес программы обработки прерывания. Затем микропроцессор 8088 начнет исполнять эту программу.

Как упоминалось в гл. 1, 256 векторов прерывания размещаются в области памяти с младшими адресами. Так как каждый из них имеет длину 4 байта, то все они занимают первые 1К байтов, т.е. область памяти с абсолютными адресами от 0 до 3FFH. Например, команда

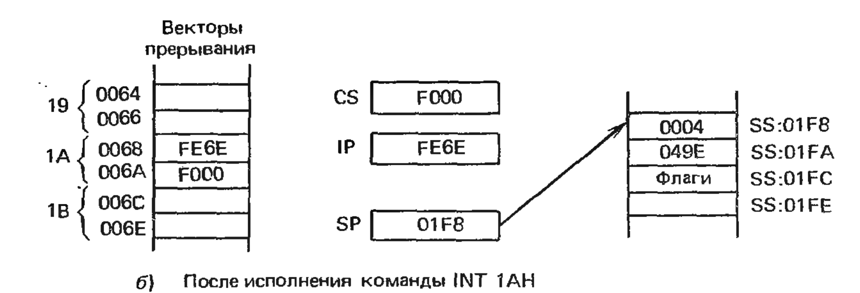
```
INT 1AH
```

заставит микропроцессор 8088 вычислить адрес вектора 68H (4*1AH). Следовательно он получит 16-битовые значения регистров IP и CS, отвечающие программе обработки прерывания, из ячеек 68H и 6AH соответственно.

На рис. 3.11 показаны стек, указатель стека SP, регистр сегмента команд CS и указатель команд IP до и после исполнения приведенной выше команды. В дан-



а) Перед исполнением команды INT 1AH



б) После исполнения команды INT 1AH

Рис. 3.11. Воздействия прерывания на стек

ном примере мы предполагаем, что вектор прерывания содержит адрес F000:FE6E. Из ячейки с этим адресом микропроцессор 8088 извлечет следующую команду.

Из 256 типов прерываний фирма Intel зарезервировала первые 5 (от 0 до 4) для внутренних прерываний. В IBM PC многие другие типы прерываний зарезервированы для нужд основной системы ввода-вывода (BIOS), а также для операционной системы DOS и интерпретатора языка Бейсик. В гл. 6 мы подробнее обсудим системные прерывания.

КОМАНДА ПРЕРЫВАНИЯ ПО ПЕРЕПОЛНЕНИЮ INTO

Команда INTO (interrupt if overflow – прервать при переполнении) представляет собой команду *условного* прерывания. Она инициирует прерывание лишь тогда, когда флаг переполнения OF равен 1. В этом случае команда INTO передает управление программе обработки прерываний с помощью косвенного вызова через вектор прерывания 4. (Другими словами, команда INTO инициирует прерывание типа 4.)

КОМАНДА ВОЗВРАТА ПОСЛЕ ПРЕРЫВАНИЯ IRET

По отношению к прерываниям команда IRET (interrupt return – возврат после обработки прерывания) играет ту же роль, что и команда RET для вызовов процедур. Она "откатывает" всю работу исходной операции и заставляет микропроцессор 8088 аккуратно выполнить возврат к основной программе. По этой причине команда IRET должна быть последней при исполнении микропроцессором 8088 программы обработки прерывания.

Команда IRET извлекает из стека три 16-битовых значения и загружает их в указатель команд IP, регистр сегмента команд CS и регистр флагов соответствен-

но. Содержимое других регистров может быть уничтожено, если в программе обработки прерывания не предусмотрено их сохранение.

3.10. КОМАНДЫ УПРАВЛЕНИЯ МИКРОПРОЦЕССОРОМ

Эти команды позволяют управлять работой микропроцессора 8088 из программы. Как показано в табл. 3.15, они делятся на три группы: команды управления флагами, команды внешней синхронизации и команда холостого хода NOP.

КОМАНДЫ УПРАВЛЕНИЯ ФЛАГАМИ

У микропроцессора 8088 есть семь команд, которые позволяют изменять флаг переноса CF, флаг направления DF и флаг прерывания IF.

Команды STC (set Carry flag – установить флаг переноса) и CLC (clear Carry flag – обнулить флаг переноса) переводят флаг CF в состояния 1 и 0 соответственно. Они полезны для установки нужного состояния флага CF перед исполнением команд циклического сдвига с флагом переноса RCL и RCR. Команда CMC (complement Carry flag – обратить флаг переноса) переводит флаг CF в состояние 0, если он имел состояние 1, и наоборот.

Таблица 3.15. **Команды управления микропроцессором**

Мнемокод	Формат	Флаги								
		OF	DF	IF	TF	SF	ZF	AF	PF	CF
Управление флагами										
STC	STC	–	–	–	–	–	–	–	–	1
CLC	CLC	–	–	–	–	–	–	–	–	0
CMC	CMC	–	–	–	–	–	–	–	–	*
STD	STD	–	1	–	–	–	–	–	–	–
CLD	CLD	–	0	–	–	–	–	–	–	–
STI	STI	–	–	1	–	–	–	–	–	–
CLI	CLI	–	–	0	–	–	–	–	–	–
Внешняя синхронизация										
HLT	HLT	–	–	–	–	–	–	–	–	–
WAIT	WAIT	–	–	–	–	–	–	–	–	–
ESC	ESC код_внеш_оп, источник	–	–	–	–	–	–	–	–	–
LOCK	LOCK	–	–	–	–	–	–	–	–	–
Холостой ход										
NOP	NOP	–	–	–	–	–	–	–	–	–

Примечание. – означает сохранение значения флага, * – его изменение.

Команда STD (set direction flag – установить флаг направления) и CLD (clear direction flag – обнулить флаг направления) переводят флаг DF в состояния 1 и 0 соответственно. Они используются для указания направления обработки строк. Если флаг DF равен 0, то после каждой операции над строкой значения индексных регистров SI и DI увеличиваются; если флаг DF равен 1, то они уменьшаются. Например, последовательность команд

```
        MOV   CX,100
        CLD
REP     MOVS DEST,SOURCE
```

пересылает элементы SOURCE,..., SOURCE+99 в ячейки DEST,..., DEST+99. А последовательность команд

```
        MOV   CX,100
        STD
REP     MOVS DEST,SOURCE
```

наоборот, пересылает элементы SOURCE,..., SOURCE–99 в ячейки DEST,..., DEST–99.

Команда CLI (clear interrupt flag – обнулить флаг прерывания) обнуляет флаг IF, что заставляет микропроцессор 8088 игнорировать маскируемые прерывания, инициируемые внешними устройствами системы. Обычно игнорирование таких прерываний требуется в тех случаях, когда микропроцессор выполняет приоритетную или критичную по времени исполнения работу, которую нельзя прерывать. Однако если флаг IF равен 0, микропроцессор все-таки будет обрабатывать немаскируемые прерывания.

Команда STI (set interrupt flag – установить флаг прерываний) переводит флаг IF в состояние 1, что разрешает микропроцессору 8088 реагировать на прерывания, инициируемые внешними устройствами.

КОМАНДЫ ВНЕШНЕЙ СИНХРОНИЗАЦИИ

Эти команды используются в основном для синхронизации действий микропроцессора 8088 с внешними событиями.

Команда HLT (halt – остановиться) переводит микропроцессор 8088 в состояние останова, при котором он находится на холостом ходу и не выполняет никакие команды. Микропроцессор 8088 выходит из состояния останова только в том случае, если Вы заново инициировали его или если он получил внешнее прерывание, немаскируемое или маскируемое (если флаг IF равен 1). Команду HLT можно использовать для перевода микропроцессора в состояние ожидания прерывания (например, набора символа на клавиатуре) перед последующей его обработкой.

Команда WAIT (wait – ожидать) также переводит микропроцессор 8088 на холостой ход, но при этом через каждые пять тактов он проверяет активность входной линии по имени TEST. В этом состоянии микропроцессор способен обрабатывать прерывания, но по завершении программы обработки прерывания он вновь возвратится в это состояние.

Если линия TEST становится активной, то микропроцессор 8088 переходит к следующей за WAIT команде. Назначение команды WAIT – останов микропроцессора на то время, пока некоторое внешнее устройство не завершит свою работу.

Прочитав, что команды HLT и WAIT останавливают микропроцессор, Вы можете предположить, что команда ESC (escape – убежать) отправит его в отпуск! Это не

так. Команда ESC попросту заставляет его извлечь содержимое указанного в ней операнда и передать его на шину данных. Тем самым команда ESC обеспечивает другим микропроцессорам системы возможность получения своих команд из потока команд микропроцессора 8088.

Команда ESC имеет формат

```
ESC внешний_код,источник
```

где **внешний_код** – 6-битовый непосредственный операнд, а **источник** – регистр или переменная. Команда ESC часто используется для передачи команд математическому сопроцессору 8087 (подробнее об этом см. в гл. 12). В этом случае **внешний_код** представляет собой код операции сопроцессора 8087, а содержимое **источник** – его операнд.

Однобайтовый префикс LOCK (lock the bus – замкнуть шину) может предшествовать любой команде. Он заставляет микропроцессор 8088 активизировать сигнал LOCK своей шины на все время исполнения этой команды. А пока сигнал LOCK активен, никакой другой процессор системы не может использовать шину.

КОМАНДА ХОЛОСТОГО ХОДА

Последняя команда NOP (no operation – нет операции) проще всех, так как она не выполняет никакой операции. Команда NOP не действует ни на флаг, ни на регистры, ни на ячейки памяти; она только увеличивает значение указателя команд IP.

Как ни странно, существует множество применений команды NOP. Например, ее кодом операции (90H) можно "забить" объектный код в том случае, если Вам надо удалить команду, не транслируя программу заново. Кроме того, команда NOP удобна при тестировании последовательности команд: можно сделать команду NOP последней в тестируемой программе и тем самым получить удобное место для остановки трассировки. Можно найти и другие приложения для этой полезной команды.

3.11. ОБЗОР КЛЮЧЕВЫХ МОМЕНТОВ ГЛАВЫ

В этой главе Вы познакомились со следующими ключевыми моментами:

1. Микропроцессор 8088 может использовать семь различных методов (или *режимов*) для получения тех данных, которыми должна оперировать команда. Ассемблер задает ему соответствующий режим на основе формата операнда, указанного в исходной программе.

2. Простейшими режимами адресации являются *регистровый* и *непосредственный*, поскольку в этом случае микропроцессор 8088 получает значение операнда из регистра или непосредственно из команды. При остальных пяти режимах адресации – *прямом, косвенном, регистровом, по базе, прямом* с *индексированием, по базе с индексированием,* микропроцессор 8088 должен вычислить адрес ячейки памяти, затем прочитать из нее значение операнда. Вычисление адреса может заключатся в простом извлечении его из команды (при прямой адресации), но может потребовать и сложения до трех компонентов (при адресации по базе с индексированием).

3. *Команды пересылки данных* микропроцессора 8088 пересылают данные и адреса между регистрами или между регистром и ячейкой памяти либо портом ввода-вывода. Команды этой группы делятся на четыре подгруппы: *команды общего назначения, команды ввода-вывода, команды пересылки адреса* и *команды пересылки флагов*.

4. Наиболее употребительная команда общего назначения MOV копирует байт или слово из регистра в ячейку памяти и наоборот, а также из регистра в регистр. Она может также скопировать непосредственное значение в регистр или ячейку памяти.

Другими полезными командами общего назначения являются команды PUSH (помещает слово в стек) и POP (извлекает слово из стека). Они позволяют сохранять в стеке содержимое регистров на то время, пока программа использует их в каких-то других целях.

5. Команды пересылки адреса содержат команду загрузки исполнительного адреса LEA, которая загружает в регистр смещение адреса ячейки памяти. Команда LEA часто применяется совместно с командами обработки строк, когда требуется найти смещение адреса каждой обрабатываемой строки.

6. У микропроцессора 8088 есть команды, выполняющие арифметические операции над двоичными числами со знаком и без знака, а также над упакованными и неупакованными десятичными числами.

7. Упакованное десятичное число содержит по две цифры в каждом байте, а неупакованные – только по одной. Каждая цифра кодируется четырьмя битами, представляющими значения от 0 до 9. Так как эти цифры десятичные, а не двоичные, то подобные числа называются двоично-десятичными, или BCD-числами.

8. Существуют две разные команды сложения. Команда сложения ADD может складывать числа, помещающиеся в одном байте или в одном слове, а также младшие биты чисел повышенной точности. А команда сложения с добавлением переноса ADC может складывать только старшие биты чисел повышенной точности. Команда приращения приемника на единицу INC добавляет 1 к операнду, но не действует на флаг переноса CF.

9. Существуют также две разные команды вычитания: команда вычитания SUB и команда вычитания с заемом SBB. Имеется также команда уменьшения приемника на единицу DEC.

10. Умножение чисел без знака осуществляется командой MUL, а чисел со знаком – командой IMUL. Обе извлекают один операнд из команды, а другой либо из регистра AL (при умножении байтов), либо из регистра AX (при умножении слов). Произведение возвращается либо в регистрах AH и AL, либо в регистрах DX и AX.

11. Деление чисел без знака осуществляется командой DIV, а деление чисел со знаком – командой IDIV. Обе извлекают делитель из операнда команды. Если делителем является байт, то делимое извлекается из регистров AH и AL, частное возвращается в регистре AL, а остаток – в регистре AH. Если делителем является слово, то делимое извлекается из регистров DX и AX, частное возвращается в регистре AX, а остаток – в регистре DX.

12. Микропроцессор 8088 при всех арифметических операциях рассматривает операнды как двоичные числа. Если Вы складываете, вычитаете и умножаете BCD-числа, то должны скорректировать результат. Для этого после команды ADD надо указать команду AAA (скорректировать сложение для представления в кодах ASCII), если числа не упакованы, или команду DAA (скорректировать

сложение для представления в десятичной форме), если числа упакованы. Аналогично после команды SUB надо указать команду AAS (скорректировать вычитание для представления в кодах ASCII) или DAS (скорректировать вычитание для представления в десятичной форме). А после команды MUL надо указать команду AAM (скорректировать умножение для представления в кодах ASCII).

Перед делением десятичных чисел надо преобразовать неупакованное делимое в двоичное число. Для этого перед командой DIV надо указать команду AAD (скорректировать деление для представления в кодах ASCII).

13. Команды расширения знака позволяют Вам оперировать смешанными данными. Команда преобразования байта в слово CBW расширяет байтовое содержимое регистра AL до слова в регистре AX. А команда преобразования слова в двойное слово расширяет слово в регистре AX до двойного слова в регистрах DX и AX.

14. *Команды манипулирования битами* микропроцессора 8088 делятся на логические команды, команды сдвига и циклического сдвига.

15. Логические команды выполняют операции AND (И), OR (ИЛИ) и XOR (исключающее ИЛИ) над двумя операндами. Команда проверки TEST (вариант команды AND) воздействует только на флаги и не изменяет операнды. Команда NOT выполняет дополнение операнда до единицы.

16. Команды сдвига смещают биты операнда влево или вправо. Для сдвига чисел со знаком надо использовать команды арифметического сдвига влево – SHL или вправо – SHR. С помощью этих команд можно быстро выполнять умножение и деление.

17. Команды циклического сдвига аналогичны командам сдвига, за исключеним того, что не уничтожают, а сохраняют биты, смещенные за пределы операнда. Команды циклического сдвига влево ROL и вправо ROR вводят эти биты в операнд с противоположного конца; команды циклического сдвига с флагом переноса влево RCL и вправо RCR помещают смещенный за пределы операнда бит во флаг CF и вводят предыдущее значение флага CF в операнд.

18. *Команды передачи управления* могут заставить микропроцессор 8088 перейти к другой части программы безусловно, при выполнении условия или при повторении блока команд – *цикла.*

19. Существуют три команды безусловной передачи управления. По команде вызова процедуры CALL микропроцессор 8088 переходит к процедуре, а по команде возврата из процедуры RET он возращается к вызвавшей ее программе. Команда перехода JMP заставляет его перейти к другой части программы.

20. Команды условной передачи управления заставляют микропроцессор 8088 осуществить переход только в случае выполнения определенного условия. Обычно им предшествует команда сравнения CMP, изменяющая состояния флагов.

21. Команды управления циклом организуют повторение его команд, используя в качестве счетчика регистр CX. Вариации LOOPE (повторять, пока равно) и LOOPNE (повторять, пока не равно) предусматривают альтернативное завершение цикла при значении флага ZF, равном 0 и 1 соответственно.

22. *Команды обработки строк* позволяют выполнять действия над последовательно расположенными байтами или словами. При выполнении этих команд микропроцессор 8088 предполагает, что операнд строка_приемник находится в дополнительном сегменте и адресуется регистром DI, а операнд строка_источник

находится в сегменте данных и адресуется регистром SI. Для выполнения операций над последовательно расположенными элементами перед командами обработки строк надо указать префикс повторения (REP, REPE или REPNE).

23. Команды обработки строк обеспечивают пересылку (MOVS), сравнение (CMPS), сканирование (SCANS), загрузку (LODS) и сохранение (STOS) строки. Каждая из них имеет три формы: первая имеет строковый операнд (или операнды), а двумя другими Вы задаете только размер элементов операндов (байты или слова).

24. Существуют три *команды прерывания*. Команда INT инициирует одно из 256 прерываний в зависимости от указанного в ней числа. Микропроцессор 8088 получает адрес программы обработки прерывания из 32-битового *вектора*, находящегося в области памяти с младшими адресами. Команда возврата после прерывания IRET заставляет микропроцессор вернуться к вызвавшей программе (подобно команде RET при вызове процедуры). Команда прерывания при переполнении INTO представляет собой условную форму команды INT; она инициирует прерывание только в том случае, когда флаг прерывания OF равен 1.

25. *Команды управления микропроцессором* позволяют управлять режимом работы микропроцессора 8088. К ним относятся команды, управляющие флагами переноса, направления и прерывания (CF, DF и IF), и команды синхронизации действий микропроцессора 8088 и другого микропроцессора. К ним принадлежит и команда холостого хода NOP, которая полезна для правки оттранслированных объектных объектных программ.

УПРАЖНЕНИЯ

1. Напишите команду, которая загружает содержимое регистра AX в ячейку памяти (размером в слово) по имени SAVE_AX, находящуюся в дополнительном сегменте.

2. Определите, что делает следующая последовательность команд:

```
MOV AX,0
MOV BX,AX
MOV BP,AX
MOV [BX],AX
MOV [BP],AX
```

3. Какие из приведенных ниже команд и фрагментов программ ошибочны? (Исходите из того, что переменные определены в сегменте данных, а команды — в сегменте команд.)

```
а)    K     EQU 1024
            ...
            ...
            MOV K,AX
б)    TEMP  DB ?
            ...
            ...
            MOV AL,TEMP
в)    TEMP  DB ?
            ...
            ...
            MOV TEMP,AX
г)    TEMP  DB ?
      T3    DB 10
            ...
            ...
            MOV TEMP,T3
```

д) MOV [BX][BP],AX

4. Укажите две команды, обнуляющие регистр AX.

5. Чем отличаются действия следующих команд:

```
MOV BX,OFFSET TABLE+4
LEA BX,TABLE+4
```

6. Напишите цикл, с помощью которого вычитается переменная V2, занимающая три слова, из переменной V1 той же длины.

7. Опишите действие следующей команды:

```
MUL 10
```

8. Если регистр AX содержит 1234H, а регистр BX — 4321H, то укажите значение регистра AX после исполнения каждой из следующих команд:

а) AND AX,BX

б) OR AX,BX

в) XOR AX,BX

г) NOT AX

д) TEST AX,BX

9. Напишите последовательность команд для *нормализации* значения регистра AX. Иначе говоря, содержимое регистра AX надо сдвигать влево до тех пор, пока старший из битов, имевших значение 1, не сместится в бит 15. Если значение регистра AX равно 0 или бит 15 уже содержит 1, то немедленно выйдите из цикла.

10. Опишите действие следующей последовательности команд:

```
START:  MOV CX,3
        SUB AX,10
        LOOP START
```

ГЛАВА 4. ОПЕРАЦИИ НАД ЧИСЛАМИ ПОВЫШЕННОЙ ТОЧНОСТИ

Если Вам приходилось программировать на языке ассемблера для 8-битового микропроцессора, то Вы должны по достоинству оценить арифметический потенциал микропроцессора 8088. Для начинающих наличие у микропроцессора 8088 встроенных команд умножения и деления означает, что часы (а то и дни), которые в противном случае пришлось бы потратить на разработку программ умножения и деления, можно провести более продуктивно, например поиграть в теннис.

В этой главе на основе команд умножения и деления мы разработаем несколько программ, которые позволят быстрее решать математические задачи. Начнем с программ умножения 32-битовых чисел, затем обсудим обработку переполнения при выполнении деления, а в конце приведем программу вычисления квадратного корня.

4.1. УМНОЖЕНИЕ

В главе 3 мы познакомились с двумя командами умножения микропроцессора 8088: с командой умножения чисел без знака MUL и командой умножения со знаком IMUL. Эти команды умножают операнды длиной в байт или слово, давая результат двойной длины (16- или 32-битовый).

Трудно ли умножать числа, занимающие более 16 битов? Как Вы скоро увидите, вовсе нет. Любой программист, которому приходилось составлять программу умножения для 8-битового микропроцессора, знает, что *наличие* команды умножения компенсирует любые неудобства, связанные с расширением ее возможностей.

УМНОЖЕНИЕ ДВУХ 32-БИТОВЫХ ЧИСЕЛ БЕЗ ЗНАКА

Команда MUL может умножать только 8- или 16-битовые значения, но ею можно воспользоваться и для умножения чисел повышенной точности без знака. Например, с ее помощью можно перемножить два 32-битовых числа. Для этого надо вычислить серию 32-битовых *перекрестных произведений,* а затем скомбинировать из них 64-битовый окончательный результат. Этот метод Вы изучали в начальной школе при умножении десятичных чисел в столбик. Как Вы помните (хотя в наше время карманных калькуляторов эти воспоминания могут оказаться довольно смутными), множитель записывается под множимым, а затем выполняется серия умножений – по одному для каждой цифры множителя. Таким образом, каждое частное произведение сдвигается на одну цифру влево по отношению к предыдущему произведению.

Умножим, например, 124 на 103:

```
    124   (множимое)
 ×  103   (множитель)
    ---
    372   (первое частное произведение)
   000    (второе частное произведение)
  124     (третье частное произведение)
  -----
  12772   (окончательный результат)
```

Сдвиг частных произведений отвечает *десятичным весам* цифр множителя. В нашем примере цифра 3 представляет единицы, цифра 0 – десятки, а цифра 1 – сотни. Следовательно, приведенный выше пример можно записать в виде

$$103 \times 124 = (3 \times 124) + (0 \times 124) + (100 \times 124)$$

или

$$103 \times 124 = (3 \times 1 \times 124) + (0 \times 10 \times 124) + (1 \times 100 \times 124).$$

В этом разделе мы разработаем короткую процедуру, которая перемножает два 32-битовых числа без знака и дает 64-битовое произведение без знака. Если бы в Вашем распоряжении не было команды умножения, то Вам пришлось бы выполнить 32 отдельных умножения (по одному для каждого бита множителя).

К счастью, у микропроцессора 8088 есть команда, непосредственно умножающая 16-битовые числа без знака. Эта команда MUL, позволяющая нам трактовать 32-битовый множитель и 32-битовое множимое как два двузначных числа, у которых каждая цифра занимает 16 битов. Таким образом мы можем получить 64-битовое произведение с помощью *четырех* умножений.

На рис. 4.1. показано символическое представление множителя (цифры А и В) и множимого (цифры С и Д) и изображено получение 64-битового окончательного результата из четырех частных произведений. Числа в кружках обозначают четыре 16-битовых сложения, которые необходимо выполнить для получения окончательного результата. (Например, сложение 1 выполняется над старшими 16 битами произведения 1 и младшими 16 битами произведения 2.)

Взяв за основу метод, представленный на этом рисунке, можно разработать процедуру умножения двух 32-битовых чисел. (Ниже в примере 4.1. показана та-

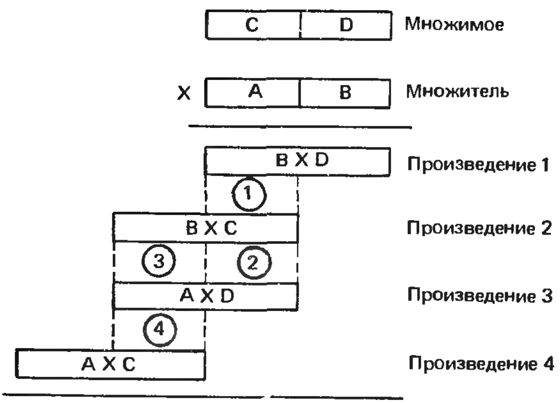


Рис. 4.1. Получение 64-битового произведения из четырех умножений пар 16-разрядных чисел

кая процедура (названная MULU32), которая получает множитель и множимое из пар регистров CX : BX и DX : AX соответственно.) Эта процедура возвращает 64-битовое произведение в тех же самых регистрах: DX (старшие 16 битов), CX (старшие 16 из средних 32 битов), BX (младшие 16 из средних 32 битов) и AX (младшие 16 битов). В примере 4.1 используется также несколько рабочих ячеек, которые должны быть определены в сегменте данных.

Если при чтении команд и комментариев процедуры MULU32 Вы будете заглядывать в рис. 4.1, то увидите, что она очень проста. Вначале процедура MULU32 сохраняет множимое в памяти, затем образует четыре 32-битовых частных произведения. После того как частные произведения сохранены в памяти, остается только сложить их. Учтите, что флаг переноса не изменяется между сложениями, поскольку команда MOV не влияет на его состояние.

Так как 32-битовый операнд без знака может принимать значения до $4{,}294 * 10^9$, то процедура MULU32 вполне пригодна для большинства приложений. (В тех случаях, когда требуются еще большие числа, надо использовать числа с плавающей точкой). Однако вполне реально разработать и процедуру умножения 64-битовых чисел (или чисел большей разрядности) по методу перекрестных произведений.

ПРИМЕР 4. 1. ПРОЦЕДУРА УМНОЖЕНИЯ ДВУХ 32-БИТОВЫХ ЧИСЕЛ БЕЗ ЗНАКА

```
; Эта процедура перемножает два 32-битовых числа без знака и
; возвращает 64-битовое произведение. Перед входом в процедуру
; множимое берется из регистров CX (старшее слово) и BX (младшее
; слово), а множитель - из регистров DX (старшее слово) и AX
; (младшее слово). Произведение возвращается в регистрах DX, CX,
; BX и AX (порядок указан от старших битов к младшим).
;
; Следующие переменные должны быть определены в сегменте данных:
;
HI_MCND   DW    ?
LO_MCND   DW    ?
HI_PP1    DW    ?
LO_PP1    DW    ?
HI_PP2    DW    ?
```

```
LO_PP2    DW    ?
HI_PP3    DW    ?
LO_PP3    DW    ?
HI_PP4    DW    ?
LO_PP4    DW    ?
;
; Основная процедура
;
MULU32    PROC
          MOV   HI_MCND,DX     ;Сохранить множимое в памяти
          MOV   LO_MCND,AX
          MUL   BX             ;Образовать частичное произведение N1
          MOV   HI_PP1,DX      ; и сохранить его в памяти.
          MOV   LO_PP1,AX
          MOV   AX,HI_MCND     ;Образовать частичное произведение N2
          MUL   BX
          MOV   HI_PP2,DX      ; и сохранить его в памяти
          MOV   LO_PP2,AX
          MOV   AX,LO_MCND     ;Образовать частичное произведение N3
          MUL   CX
          MOV   HI_PP3,DX      ; и сохранить его в памяти
          MOV   LO_PP3,AX
          MOV   AX,HI_MCND     ;Образовать частичное произведение N4
          MUL   CX
          MOV   HI_PP4,DX      ; и сохранить его в памяти
          MOV   LO_PP4,AX
;
; Сложить частичные произведения для получения полного 64-битового
; произведения
;
          MOV AX,LO_PP1        ;Младшие 16 битов
          MOV BX,HI_PP1        ;Сформировать младшую часть средних битов
          ADD BX,LO_PP2        ; с суммой N1
          ADC HI_PP2,0
          ADD BX,LO_PP3        ; и суммой N2
          MOV CX,HI_PP2        ;Сформировать старшую часть средних битов
          ADC CX,HI_PP3        ; с суммой N3
          ADC HI_PP4,0
          ADD CX,LO_PP4        ; и суммой N4
          MOV DX,HI_PP4        ;Сформировать старшие 16 битов
          ADC DX,0             ; с продвинутым флагом переноса
          RET
MULU32    ENDP
```

УМНОЖЕНИЕ ДВУХ 32-БИТОВЫХ ЧИСЕЛ СО ЗНАКОМ

Процедура умножения чисел без знака (пример 4.1) может умножать и числа со знаком, если они положительны. Другими словами, этот пример представляет собой процедуру умножения двух 32-битовых неотрицательных чисел.

С помощью этой процедуры нельзя умножать отрицательные числа, потому что они представляются в форме *дополнения до двух*. Ну хорошо, а если просто заменить в примере 4.1 каждую команду MUL на команду умножения чисел со знаком IMUL? К несчастью, это не поможет, потому что команда IMUL рассматривает старший бит каждого операнда как признак знака, а при получении перекрестных произведений это не так.

Но как же перемножить 32-битовые числа со знаком? Один из вариантов состоит в изменении знака отрицательных операндов, выполнении обычного умножения чисел без знака и последующей коррекции знака произведения (если это необходимо). Когда только один из операндов отрицателен, надо вычислить дополнение произведения до двух. Если оба операнда отрицательны, то (положительное) произведение не требует коррекции.

Этот простой прием реализован в приведенном ниже примере 4.2, в котором байтовая переменная NEG_IND содержит признак знака "–". Изначально переменная NEG_IND полагается равной нулю и сохраняет это значение, если оба операнда положительны. Когда один из операндов отрицателен, мы выполняем дополнение переменной NEG_IND до единицы, в результате чего все ее биты станут единичными. Если оба операнда отрицательны, то мы *дважды* выполним дополнение переменной NEG_IND до единицы, в результате чего все ее биты снова станут нулевыми.

ПРИМЕР 4.2. ПРОЦЕДУРА УМНОЖЕНИЯ ДВУХ 32-БИТОВЫХ ЧИСЕЛ СО ЗНАКОМ

```
; Эта процедура перемножает два 32-битовых числа со знаком и
; возвращает 64-битовое произведение. Перед входом в процедуру
; множимое берется из регистров CX (старшее слово) и BX (младшее
; слово), а множитель - из регистров DX (старшее слово) и AX
; (младшее слово). Произведение возвращается в регистрах DX, CX,
; BX и AX (порядок указан от старших битов к младшим)
; Эта процедура вызывает процедуру MULU32 (пример 4.1)
;
; Следующая переменная должна быть определена в сегменте данных:
;
NEG_IND  DB     ?
;
; Основная процедура
;
EXTRN    MULU32:FAR          ;MULU32 - внешняя процедура
MULS32   PROC
         MOV   NEG_IND,0     ;Индикатор знака = 0
         CMP   DX,0          ;Множимое отрицательно?
         JNS   CHKCX         ; Нет. Проверить множитель
         NOT   AX            ; Да. Выполнить дополнение множимого
         NOT   DX            ;    до 2
         ADD   AX,1
         ADC   DX,0
         NOT   NEG_IND       ;    и дополнение индикатора знака до 1
CHKCX:   CMP   CX,0          ;Множитель отрицателен?
         JNS   GOMUL         ; Нет. Перейти к умножению
         NOT   BX            ; Да. Выполнить дополнение множителя
         NOT   CX            ;    до 2
         ADD   BX,1
         ADC   CX,0
         NOT   NEG_IND       ;    и дополнение индикатора знака до 1
GOMUL:   CALL  MULU32        ;Выполнить умножение без знака
         CMP   NEG_IND,0     ;Знак произведения правилен?
         JZ    DONE          ; Да. Перейти к выходу из процедуры
         NOT   AX            ; Нет. Выполнить дополнение произведения
         NOT   BX            ;    до 2
         NOT   CX
         NOT   DX
         ADD   AX,1
         ADC   BX,0
         ADC   CX,0
         ADC   DX,0
DONE:    RET
MULS32   ENDP
```

При каждом дополнении переменной NEG_IND до единицы надо обратить знак у одного из операндов (выполнить его дополнение до двух). Так как команда NEG оперирует только байтами или словами, то нам приходится выполнять дополнение 32-битовых операндов грубым способом: сначала получать дополнение операнда до единицы, а затем добавлять к нему 1.

Для выполнения умножения двух 32-битовых чисел процедура MULS32 вызывает процедуру MULU32. Так как процедура MULU32 находится в другом исход-

Таблица 4.1. Время исполнения процедуры MULS32

Операнды	Максимальное время	
	Такты	Микросекунды
Оба положительны	1096	230,16
Противоположного знака	1136	238,56
Оба отрицательны	1144	240,24

ном модуле, то мы должны объявить ее внешней с помощью оператора EXTRN в самом начале примера. (Напомним, что в этом случае модуль MULU32 должен иметь оператор PUBLIC MULU32). По возвращении из MULU32 состояние переменной NEG_IND показывает, является ли произведение правильным (переменная NEG_IND равна 0) или оно нуждается в изменении знака (переменная NEG_IND отлична от 0).

Время исполнения процедуры MULS32 зависит от того, будут ли операнды оба положительны, оба отрицательны или противоположного знака (табл. 4.1).

4.2. ДЕЛЕНИЕ

Существует много приложений, где требуется деление; самое распространенное из них – вычисление среднего значения для ряда чисел (например, результатов лабораторных исследований). Ниже (в примере 4.3) показана типичная процедура усреднения.

Эта процедура (AVERAGE) усредняет заданное число значений без знака (размером в слово), на которые указывает регистр BX;счетчиком значений служит регистр CX. Она возвращает целую часть среднего значения в регистре AX, а дробный остаток – в регистре DX. Например, для вычисления среднего значения в таблице TABLE, состоящей из 100 слов, можно воспользоваться следующей последовательностью команд:

```
LEA BX,TABLE        ;Загрузить смещение адреса таблицы TABLE
MOV CX,100          ;и число ее элементов
CALL AVERAGE        ;Вычислить среднее значение
```

При описании команд DIV и IDIV в разд. 3.5 мы упоминали, что операция деления автоматически обрывается, если делитель равен нулю и возникло *переполнение*. Переполнение возникает только в том случае, если делимое настолько больше делителя, что частное не может поместиться в регистрах, предназначенных для возвращения результата. Деление чисел без знака приводит к переполнению, если делимое более чем в 65 535 раз превышает делитель.

Как деление на нуль, так и переполнение при делении заставляют микропроцессор 8088 инициировать прерывание типа 0 (деление на нуль). При обработке этого прерывания система сохраняет содержимое регистров, выдает сообщение

```
Divide Overflow
```

и затем аварийно завершает выполнение программы.

Используемая в примере 4.3 операция деления будет прервана, если на входе регистр CX содержит нуль. А может ли она быть прервана из-за переполнения? Нет, в данном случае этого не может быть, потому что отношение делимого (суммы слов) к делителю (счетчику слов) никогда не может превысить 65 536! Однако в других случаях (например, при делении 200 000 на 2) переполнение вполне может возникнуть. Поэтому в следующем подразделе мы рассмотрим процедуру, которая всегда возвращает правильное частное и избегает переполнения.

ПРИМЕР 4.3. ПРОГРАММА УСРЕДНЕНИЯ СЛОВ

```
; Эта процедура вычисляет среднее значение для заданного количества
; чисел без знака (размером в слово), находящихся в сегменте данных
; Смещение адреса первого слова берется из регистра BX, счетчик
; слов - из регистра CX
; Процедура возвращает целую часть среднего значения в регистре AX, а
; дробную часть (в виде остатка) в регистре DX
;
AVERAGE  PROC
         SUB    AX,AX        ;Присвоить делимому нулевое
         SUB    DX,DX        ; начальное значение
         PUSH   CX           ;Сохранить счетчик слов в стеке
ADD_W:   ADD    AX,[BX]      ;Добавить текущее слово к сумме
         ADC    DX,0
         ADD    BX,2         ; и увеличить индекс
         LOOP   ADD_W        ;Все ли слова просуммированы?
         POP    CX           ; Да. Извлечь счетчик слов
         DIV    CX           ;  и вычислить среднее значение
         RET
AVERAGE  ENDP
```

ОБРАБОТКА ПЕРЕПОЛНЕНИЯ

В некоторых ситуациях переполнение является признаком ошибки. В других случаях оно допустимо, но означает, что программа должна быть в состоянии воспринять частное, занимающее более 16 битов. Но поскольку микропроцессор 8088 прерывает деление при обнаружении переполнения, то как можно получить такое частное? Самый простой способ состоит в том, чтобы расщепить 32-битовое делимое на два 16-битовых числа, а затем выполнить два деления двух 16-битовых чисел (которые *не могут* вызвать переполнение).

Если обозначить 16-битовый делитель через X, а 32-битовое делимое – через $Y_1 Y_0$, то операцию деления можно записать в виде

$$\frac{Y_1 Y_0}{X}.$$

Если частное записать в виде двух 16-битовых цифр Q_1 и Q_0 и остаток в виде двух 16-битовых цифр R_1 и R_0, то это деление можно представить в следующем виде:

$$Q_1 * 2^{16} = \frac{Y_1 * 2^{16}}{X} \quad \text{с остатком } R_1 * 2^{16}$$

и

$$Q_0 = \frac{R_1 * 2^{16} + Y_0}{X} \quad \text{с остатком } R_0.$$

Как видите, сочетание этих двух операций дает 32-битовое частное $Q_1 Q_0$ и 16-битовый остаток R_0. (Промежуточный остаток R_1, если он есть, исчезает при выполнении второй операции деления). Если переполнения нет, то Q_1 равно 0 и результат возвращается как $0Q_0$ и R_0.

Мы можем применить это способ к разработке процедуры, которая *всегда* возвращает правильные частное и остаток и сама обрабатывает переполнение. Эта процедура, названная DIVUO, приводится в примере 4.4. Она делит 32-битовое частное, образованное значениями регистров DX (старшее слово) и AX (младшее слово), на 16-битовый делитель в регистре BX и возвращает 32-битовое частное в паре регистров BX:AX и 16-битовый остаток в регистре DX.

Процедура DIVUO состоит из четырех шагов:

1. Проверить, не равен ли нулю делитель в регистре BX. Если равен, то вызвать прерывание типа 0 для прекращения операции.

2. Изменить вектор прерывания типа 0, находящийся в ячейке с абсолютным адресом 0 (смещение) и 2 (номер блока), так, чтобы он указывал на новую процедуру обработки прерывания, имеющую в примере 4.4 метку OVR_INT.

3. Выполнить деление. При отсутствии переполнения микропроцессор 8088 продолжает исполнение со следующей команды (SUB BX,BX). При переполнении он вызывает процедуру обработки прерывания типа 0, в качестве которой теперь используется OVR_INT.

4. Восстановить исходный вектор прерывания типа 0 по значениям, ранее помещенным в стек.

ПРИМЕР 4.4. ПРОЦЕДУРА ДЕЛЕНИЯ С ОБРАБОТКОЙ ПЕРЕПОЛНЕНИЯ

```
; Эта процедура деления возвращает корректные частное и остаток
; даже в случае возникновения переполнения; 16-битовый делитель
; берется из регистра BX, а 32-битовое делимое - из регистров
; DX (старшее слово) и AX (младшее слово)
; 32-битовое частное возвращается в регистрах BX:AX, а 16-битовый
; остаток - в регистре DX
;
DIVUO     PROC
          CMP   BX,0               ;Делитель = 0?
          JNZ   DVROK
          INT   0                  ; Да. Прервать деление
DVROK:    PUSH  ES                 ;Сохранить текущие значения
          PUSH  DI                 ; регистров ES, DI
          PUSH  CX                 ; и CX
          MOV   DI,0               ;Извлечь текущий вектор INT 0
          MOV   ES,DI
          PUSH  ES:[DI]            ; и сохранить его в стеке
          PUSH  ES:[DI+2]
          LEA   CX,OVR_INT         ;Сделать вектор INT 0
          MOV   ES:[DI],CX         ; указателем на метку OVR_INT
          MOV   CX,SEG OVR_INT
          MOV   ES:[DI+2],CX
          DIV   BX                 ;Выполнить деление
          SUB   BX,BX              ;Если переполнения нет, обнулить BX
RESTORE:  POP   ES:[DI+2]          ;Восстановить вектор INT 0
          POP   ES:[DI]
          POP   CX                 ;Восстановить исходные значения
          POP   DI                 ; регистров CX, DI
          POP   ES                 ; и ES
          RET
;
; Эта процедура обработки прерывания выполняется, если при
; выполнении деления возникает переполнение
;

OVR_INT:  POP   CX                 ;Изменить смещение адреса возврата
          LEA   CX,RESTORE         ; для обхода команды SUB BX,BX
          PUSH  CX
          PUSH  AX                 ;Сохранить текущее значение AX
          MOV   AX,DX              ;Установить первое делимое, 0-Y1
```

```
        SUB     DX,[illegible]X
        DIV     BX          ;Теперь (AX) = Q1, (DX) = R1
        POP     CX          ;Извлечь в CX исходное значение AX
        PUSH    AX          ;Сохранить Q1 в стеке
        MOV     AX,CX       ;Установить второе делимое, R1-Y0
        DIV     BX          ;Теперь (AX = Q0, (DX) = R0
        POP     BX          ;Окончательное частное - в BX:AX
        IRET
DIVUO   ENDP
```

Независимо от возникновения или отсутствия переполнения процедура DIVUO возвращает 32-битовое частное (BX:AX) и 16-битовый остаток. Если переполнения не было, то значение регистра BX равно нулю.

4.3. ИЗВЛЕЧЕНИЕ КВАДРАТНОГО КОРНЯ

В этом заключительном разделе мы разработаем программу вычисления целой части квадратного корня из 32-битового числа без знака. Эта программа основана на применении классического метода последовательных приближений, известного под названием метода Ньютона. Согласно этому методу *приближенное значение A квадратного корня из числа N можно улучшить с помощью формулы*

A1 = (N/A + A)/2.

Проиллюстрируем это на примере. Предположим, что Вам требуется получить квадратный корень из числа, имеющего значение N. В качестве первого приближения возьмем значение (N/200) + 2. Чтобы получить второе приближение, поделим N на первое приближение, а затем усредним два этих значения. Для получения третьего приближения поделим N на второе приближение и снова получим среднее значение и т. д. Например, при вычислении квадратного корня из 10 000 нам понадобится выполнить следующие действия:

N = 10 000; первое приближение равно (10 000/200) + 2, или 52.
10 000/52 = 192,   (192 + 52)/2 = 122,
10 000/122 = 81,   (122 + 81)/2 = 101,
10 000/101 = 99,   (101 + 99)/2 = 100,
10 000/100 = 100.

Таким образом, квадратный корень из 10 000 равен 100. Мы определяем, что квадратным корнем является именно 100, а не другие промежуточные приближения потому, что произведение 100 на 100 равно 10 000.

Выбранное нами число 10 000 имело целый квадратный корень, но далеко не все числа таковы. Например, квадратный корень из 9999 не является целым числом. Таким образом, если мы попробуем запрограммировать метод последовательных приближений и используем соотношение

корень * корень = число

в качестве критерия завершения вычислений, то процессор будет повторять команды вычисления приближений до бесконечности, поскольку квадрат *целого* приближения никогда не может стать равным 9999. Конечно, лучше всего остановить микропроцессор после того, как он найдет ближайший, или "лучший", квадратный корень из числа.

Можно пользоваться несколькими различными методами для завершения процедуры вычисления приближений. Какой из них подойдет Вам больше всего, зависит от желаемой точности ответа и ограничений на время его вычисления.

Например, можно позволить микропроцессору 8088 выполнить цикл вычисления приближения 10 раз, предполагая, что ответ будет достаточно точным. Хотя во многих приложениях этот способ вполне приемлем, но, в общем-то, он довольно произволен. Для получения более точного решения можно позволить микропроцессору 8088 повторять цикл, пока два последовательных приближения не станут идентичны или будут отличаться всего на 1. В нашем примере мы остановимся на этом способе.

В примере 4.5 показана процедура SQRT32, которая использует метод последовательных приближений для вычисления квадратного корня из 32-битового числа. Она получает исходное число из регистров DX (старшее слово) и AX (младшее слово) и возвращает 16-битовый квадратный корень в регистре BX.

Сначала процедура сохраняет значения регистров BP, DX и AX в стеке, затем копирует указатель стека SP в регистр BP. Теперь регистр BP указывает на значение регистра AX, помещенное в стек. После этого вычисляется первое приближение по формуле (N/200) + 2.

Команда с меткой NXT_APP открывает цикл, заканчивающийся меткой DONE. При каждом проходе этого цикла микропроцессор 8088 вычисляет новое приближение с помощью деления 32-битового исходного числа (считываемого из стека) на предшествующее приближение, а затем усреднения двух результатов. При усреднении значение регистра AX сдвигается вправо на один бит, что соответствует делению на два. Применение команды SHR вместо команды DIV сохраняет немало времени (команда SHR исполняется за 2 такта, а DIV – минимум за 80 тактов!).

ПРИМЕР 4.5. ИЗВЛЕЧЕНИЕ КВАДРАТНОГО КОРНЯ ИЗ 32-БИТОВОГО ЧИСЛА

```
; Эта процедура извлекает квадратный корень из 32-битового целого
; числа, которое берется из регистров DX (старшее слово) и AX
; (младшее слово), и возвращает этот квадратный корень как 16-би-
; товое целое число в регистре BX. Исходное число в регистрах
; DX:AX не изменяется
;
SQRT32  PROC
        PUSH  BP          ;Сохранить содержимое регистра BP
        PUSH  DX          ; и исходное число в регистрах DX:AX
        PUSH  AX
        MOV   BP,SP       ;Переместить BP на значение AX в стеке
        MOV   BX,200      ;В качестве начального приближения
        DIV   BX          ; разделить исходное число на 200,
        ADD   AX,2        ; затем добавить 2
NXT_APP: MOV  BX,AX       ;Сохранить полученное приближение в регистре BX
        MOV   AX,[BP]     ;Прочитать исходное число заново
        MOV   DX,[BP+2]
        DIV   BX          ;Разделить его на последнее приближение
        ADD   AX,BX       ;Усреднить два последних результата
        SHR   AX,1
        CMP   AX,BX       ;Два последних приближения идентичны?
        JE    DONE
        SUB   BX,AX       ; Нет. Сравнить их разность с +1 или -1
        CMP   BX,1
        JE    DONE
        CMP   BX,-1
        JNE   NXT_APP
DONE:   MOV   BX,AX       ;Поместить результат в регистр BX
        POP   AX          ;Восстановить исходное число
        POP   DX
        POP   BP          ; и рабочий регистр BP
        RET
SQRT32  ENDP
```

Процедура сравнивает каждое новое приближение с предыдущим, чтобы определить момент, когда они станут одинаковы или будут отличаться всего лишь на 1(+ 1 или − 1). Обнаружив это, микропроцессор передает управление метке DONE; в противном случае он возвращает управление метке NXT_APP для вычисления нового приближения. После выхода на метку DONE микропроцессор 8088 помещает окончательное значение квадратного корня в регистр BX, а затем извлекает из стека исходное число (в регистры AX и DX) и первоначальное значение регистра BP.

УПРАЖНЕНИЯ

1. Следующее интересное наблюдение, сделанное несколько лет назад, приводит нас к простому методу вычисления квадратного корня: *квадратный корень из целого числа равен количеству последовательных нечетных чисел, которое можно из него вычесть.*

Например, извлечем этим методом квадратный корень из 25. (Скептики могут проверить его на нескольких других примерах.) В нашем случае из 25 можно вычесть всего пять нечетных чисел — 1, 3, 5, 7 и 9, так что квадратный корень равен 5:

```
  25
 − 1   (приближенное значение квадратного корня равно 1)
 ───
  24
 − 3   (приближенное значение квадратного корня равно 2)
 ───
  21
 − 5   (приближенное значение квадратного корня равно 3)
 ───
  16
 − 7   (приближенное значение квадратного корня равно 4)
 ───
   9
 − 9   (квадратный корень равен 5)
 ───
   0
```

Разработайте процедуру, которая по этому методу находит квадратный корень из 32-битового числа без знака, помещенного в регистры DX (старшее слово) и AX (младшее слово), и возвращает 16-битовый квадратный корень в регистре BX. Как и в примере 4.5, значения регистров AX и DX должны быть сохранены.

ГЛАВА 5. МАНИПУЛИРОВАНИЕ СТРУКТУРАМИ ДАННЫХ

Существует почти столько же способов организации хранения информации в памяти, сколько имеется видов организуемой информации. В зависимости от приложений выделяются такие объекты, как списки, массивы, строки и табличные функции. Все это суть различные виды *структур данных*.

О структурах данных можно написать (что и делается) много томов, и мы не собираемся в этой книге давать исчерпывающее изложение этих понятий. Вместо этого мы. сосредоточимся на трех основных структурах: *списках, табличных функциях* и *текстовых файлах*.

Списки состоят из последовательно размещенных в памяти единиц данных (групп байтов или слов), называемых *элементами*. Размещение может быть *смежным*, когда элементы располагаются в соседних ячейках памяти, или *связанным*, когда каждый элемент содержит *указатель* на следующий элемент списка. Кроме того, элементы могут быть размещены в произвольном порядке или в порядке возрастания либо убывания.

Табличные функции являются структурами данных, содержащими информацию (данные или адреса), имеющую определенную связь с известным значением.

Например, телефонный справочник является табличной функцией: зная фамилию, в нем можно найти соответствующий телефонный номер.

Текстовые файлы содержат нечисловую информацию, например, письма, отчеты и списки телефонов.

5.1. НЕУПОРЯДОЧЕННЫЕ СПИСКИ

В нашем организованном обществе, где строки телефонных справочников упорядочены по алфавиту, а номера домов возрастают или убывают, если идти по улице, что-нибудь неупорядоченное может показаться неудобным. Но еще не все можно аккуратно упорядочить, и поэтому неупорядоченные списки остаются жизненно важными во многих приложениях, особенно в тех, где данные случайны или изменяются со временем. Например, ЭВМ автоматических метеостанций могут запоминать ежечасные показания термометров в неупорядоченных списках, а промышленники могут запоминать в таких списках ежемесячную статистику сбыта товаров.

Большинство списков содержат байт (или слово) счетчика элементов и один или несколько элементов данных. При работе со списком обычно требуется добавлять или удалять элементы, либо просматривать их в поисках определенного значения. Эти операции легко выполнить следующим образом:

1. Для добавления элемента надо запомнить его в конце списка и увеличить счетчик элементов на 1.
2. Для удаления элемента надо сместить в памяти последующие элементы вниз, а затем вычесть 1 из счетчика элементов.
3. Для поиска заданного элемента данных надо сравнить каждый элемент списка, начиная с первого, с искомым значением.

ДОБАВЛЕНИЕ ЭЛЕМЕНТА К НЕУПОРЯДОЧЕННОМУ СПИСКУ

Показанная в примере 5.1 процедура ADD_TO_UL представляет собой такую программу, которая используется для создания неупорядоченного списка или для добавления элемента к уже существующему списку. В данном примере список содержит значения (со знаком или без знака) размером в слово.

Процедура ADD_TO_UL считывает счетчик элементов в регистр CX, а затем сканирует данные в поисках элемента, совпадающего со значением регистра AX. Если это значение уже находится в списке (признаком этого служит нулевое конечное значение флага ZF), то микропроцессор 8088 помещает начальный адрес обратно в регистр DI и возвращается из процедуры. В противном случае он добавляет значение в конец списка и увеличивает счетчик элементов на единицу.

Как долго исполняется эта процедура? Это зависит от числа элементов и от того, находится ли искомое значение в списке. Основным фактором является число повторений команды сканирования строки SCASW. Ее исполнение занимает (9 + 15N) тактов, где N – число повторений. Попробуем оценить время исполнения процедуры в обоих случаях (значение или находится в списке, или отсутствует) для списка из N элементов.

Если искомого значения в списке нет, то команда SCASW исполняется N раз. Остальные команды процедуры исполняются только по одному разу и в сумме занимают 135 тактов. Следовательно,

время исполнения процедуры = 9 + 19N + 135 = 19N + 144 тактов.

Таким образом, добавление элемента к списку из 100 элементов займет 2044 такта или 429,24 мкс.

ПРИМЕР 5.1. ДОБАВЛЕНИЕ ЭЛЕМЕНТА К НЕУПОРЯДОЧЕННОМУ СПИСКУ

```
; Эта процедура добавляет значение, находящееся в регистре AX, к
; неупорядоченному списку в дополнительном сегменте (при условии,
; что такого значение в списке еще нет). Начальный адрес списка
; берется из регистра DI. Длина списка (в словах) находится в
; первой ячейке списка. По возвращению из процедуры значения
; регистров DI и AX не изменяются
;
ADD_TO_UL   PROC
            CLD                      ;Положить флаг DF = 0 для сканирования
;                                      слева направо
            PUSH    DI               ;Сохранить начальный адрес
            PUSH    CX
            MOV     CX,ES:[DI]       ;Извлечь счетчик слов
            ADD     DI,2
REPNE       SCASW                    ;Значение уже находится в списке?
            POP     CX
            JNE     ADD_IT
            POP     DI               ; Да. Восстановить начальный адрес
            RET                      ;  и выйти из подпрограммы
ADD_IT:     STOSW                    ; Нет. Добавить его в конец списка
            POP     DI.              ;  и увеличить счетчик слов
            INC     WORD PTR ES:[DI]
            RET
ADD_TO_UL   ENDP.
```

Если же искомое значение уже находится в списке, то в среднем его поиск потребует от микропроцессора 8088 выполнения N/2 сравнений, поскольку в половине случаев искомое значение должно быть в первой половине списка. Таким образом, теоретически поиск занимает в среднем (9 + 9,5N) тактов. Остальные команды занимают еще 78 тактов. Поэтому

среднее время исполнения = 9 + 9,5N + 78 = 9,5N + 87 тактов.

Следовательно, поиск элемента в неупорядоченном списке из 100 элементов займет в среднем 1037 тактов или 217,77 мкс.

УДАЛЕНИЕ ЭЛЕМЕНТА ИЗ НЕУПОРЯДОЧЕННОГО СПИСКА

Для удаления элемента из неупорядоченного списка Вам надо сначала найти этот элемент, а затем сдвинуть последующие элементы в памяти на одну позицию вниз[1]. Тем самым Вы перезапишите уничтожаемую "жертву". Так как элемент удален, то надо уменьшить значение счетчика элементов (первую ячейку списка) на 1.

На рис. 5.1 показано размещение списка байтов в памяти. Так как в списке шесть элементов данных, то первая ячейка (LIST) содержит значение 6. На этом рисунке показано также, как выглядит список после удаления четвертого элемента (14). В списке остались только пять элементов данных, а значения 97 и 8 передвинулись в памяти вниз, уничтожая удаляемое значение.

[1] На рис. 5.1 сдвиг происходит в противоположном направлении из-за выбранного в нем направления возрастания адресов памяти. — *Прим. перев.*

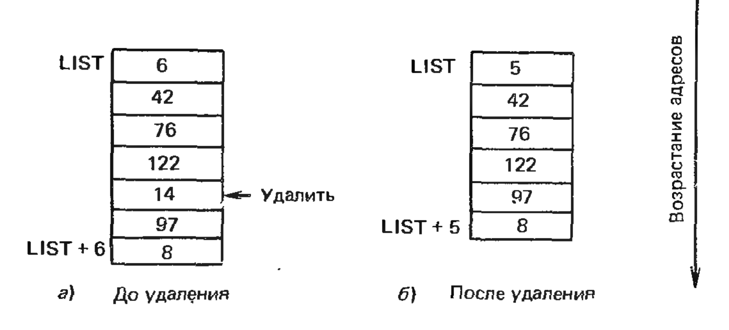


Рис. 5.1. Изменение списка при удалении элемента

Процедура DEL_UL в примере 5.2 выполняет именно эту операцию в предположении, что в регистре AX задано удаляемое значение. Как и в примере 5.1, регистр DI указывает на начало списка.

Команды, предшествующие команде REPNE, загружают счетчик элементов в регистр CX, а адрес первого элемента данных – в регистр DI. После этого список сканируется в поисках заданного значения. Эти команды идентичны командам начала примера 5.1. Если искомое значение находится в списке (ZF = 1), то микропроцессор 8088 переходит к метке DELETE.

В этом месте микропроцессор выбирает один из двух путей. Если удаляемый элемент находится в самом конце списка (регистр CX содержит нуль), то микропроцессор 8088 переходит к метке DEC_INT, где происходит просто уменьшение счетчика элементов списка. Если же удаляемый элемент находится в каком-то другом месте списка, то цикл, начинающийся с метки NEXT_EL, переместит все последующие элементы на одну позицию вниз, перезаписывая "жертву". Затем счетчик элементов уменьшается на 1, чтобы зафиксировать удаление.

ПРИМЕР 5.2. УДАЛЕНИЕ ЭЛЕМЕНТА ИЗ НЕУПОРЯДОЧЕННОГО СПИСКА

```
; Эта процедура удаляет значение, находящееся в регистре AX, из
; неупорядоченного списка в дополнительном сегменте (при условии,
; что такое значение в списке есть). Начальный адрес списка
; берется из регистра DI. Длина списка (в словах) находится в
; первой ячейке списка. По возвращению из процедуры значения
; регистров DI и AX не изменяются
;
DEL_UL    PROC
          CLD                  ;Положить флаг DF = 0 для сканирования
;                                слева направо
          PUSH   BX            ;Сохранить рабочий регистр BX
          PUSH   DI            ; и начальный адрес
          MOV    CX,ES:[DI]    ;Извлечь счетчик слов
          ADD    DI,2
REPNE     SCASW                ;Значение уже находится в списке?
          JE     DELETE        ; Да. Удалить его
          POP    DI            ; Нет. Восстановить регистры
          POP    BX
          RET                  ;  и выйти из процедуры
;
; Следующие команды удаляют элемент из списка по следующему
; алгоритму:
```

```
;    (1) Если элемент находится в конце списка, удалить его
;        путем уменьшения счетчика элементов на 1
;    (2) В противном случае удалить его путем сдвига всех
;        следующих за ним элементов на одну позицию влево
;
DELETE:     JCXZ  DEC_CNT      ;Если (CX) = 0, удалить последний
;                                элемент

NEXT EL:    MOV   BX,ES:[DI]   ;Сдвинуть один элемент списка влево
            MOV   ES:[DI-2],BX
            ADD   DI,2         ;Указать на следующий элемент
            LOOP  NET_EL       ;Повторять, пока не будут сдвинуты
;                                все элементы
DEC_CNT:    POP   DI           ;Уменьшить счетчик элементов на 1
            DEC   WORD PTR ES:[DI]
            POP   BX           ;Восстановить содержимое регистра BX
            RET                ; и выйти из процедуры
DEL_UL      ENDP
```

ПОИСК МАКСИМУМА И МИНИМУМА В НЕУПОРЯДОЧЕННОМ СПИСКЕ

Иногда бывает необходимо найти в неупорядоченном списке наибольшее и наименьшее значения. Один из способов решения этой задачи состоит в том, что первоначально максимальное и минимальное значения полагаются равными первому элементу, а затем с ними сравниваются остальные элементы списка. Если программа находит элемент, значение которого меньше текущего минимума, то она считает его новым *минимумом*. Аналогично если она находит элемент, значение которого больше текущего максимума, то она считает его новым *максимумом*.

В процедуре MINMAX из примера 5.3 этот метод применен к неупорядоченному списку слов без знака, начальный адрес которого находится в регистре DI. Процедура MINMAX возвращает максимальное и минимальное значения в регистрах AX и BX соответственно.

Эта процедура состоит из трех частей. Первая часть определяет требуемое число сравнений (равное числу элементов списка без единицы) и полагает минимум и максимум равными значению первого элемента. Вторая и третья части представляют собой цикл просмотра списка в поисках новых значений минимума и максимума соответственно. Новые значения минимума загружаются в регистр BX, а максимума – в регистр AX.

Хотя процедура MINMAX рассчитана на обработку списка слов без знака, из нее нетрудно получить процедуру поиска максимума и минимума в списках слов со знаком. Для этого в примере 5.3 достаточно заменить команду JAE NOMIN на команду JGE NOMIN, а команду JBE NOMAX на команду JLE NOMAX (см. табл. 3.11 в разд. 3.7).

ПРИМЕР 5.3. ПОИСК МАКСИМАЛЬНОГО И МИНИМАЛЬНОГО ЗНАЧЕНИЙ В НЕУПОРЯДОЧЕННОМ СПИСКЕ

```
; Эта процедура находит максимальное и минимальное значение слов
; неупорядоченного списка в дополнительном сегменте и возвращает
; эти значения в регистрах AX и BX соответственно
; Начальный адрес списка берется из регистра DI. Длина списка
;(в словах) находится в первой ячейке списка. По возвращению из
; процедуры значение регистра DI не изменяется
;
```

```
MINMAX   PROC
         PUSH  CX
         PUSH  DI               ;Сохранить начальный адрес
         MOV   CX,ES:[DI]       ;Извлечь счетчик слов
         DEC   CX               ;Подготовиться к (счетчик-1) сравнениям
         PUSH  CX               ;Сохранить текущее значение счетчика
         MOV   BX,ES:[DI+2]     ;Сначала сделаем максимум и минимум
         MOV   AX,BX            ; равными значению первого элемента
;
; Эти команды находят в списке минимальное значение
;
         ADD   DI,4             ;Указать на второй элемент списка
         PUSH  DI               ; и сохранить этот указатель
CHKMIN:  CMP   ES:[DI],BX       ;Сравнить следующий элемент с минимумом
         JAE   NOMIN            ;Найден новый минимум?
         MOV   BX,ES:[DI]       ; Да. Поместить его в регистр BX
NOMIN:   ADD   DI,2             ;Указать на следующий элемент
         LOOP  CHKMIN           ;Проверить весь список
;
; Эти команды находят в списке максимальное значение
;
         POP   DI               ;Указать на второй элемент списка
         POP   CX               ;Перезагрузить счетчик числа сравнений
CHKMAX:  CMP   ES:[DI],AX       ;Сравнить следующий элемент с максимумом
         JBE   NOMAX            ;Найден новый максимум?
         MOV   AX,ES:[DI]       ; Да. Поместить его в регистр AX
NOMAX:   ADD   DI,2             ;Указать на следующий элемент
         LOOP  CHKMAX           ;Проверить весь список
         POP   DI
         POP   CX
         RET
MINMAX   ENDP
```

5.2. СОРТИРОВКА НЕУПОРЯДОЧЕННЫХ ДАННЫХ

Если Вы изображаете график изменения данных во времени или обрабатываете текст, то неупорядоченные (по значению) данные оказываются вполне приемлемыми. Однако во многих приложениях информацию легче анализировать, если она упорядочена по возрастанию или убыванию.

Каким образом можно изменить расположение данных в неупорядоченном списке? На эту тему опубликовано много литературы, но мы ограничимся изложением одного из общеупотребительных методов, известного под названием *пузырьковой сортировки*. Если Вы хотите познакомиться с другими методами сортировки, то лучше всего начать с классики – книги Д. Кнута "Искусство программирования. Т. 3. Сортировка и поиск": Пер. с англ. – М.: – Мир, 1978. – 846 с.

ПУЗЫРЬКОВАЯ СОРТИРОВКА

Метод пузырьковой сортировки назван так потому, что при его применении элементы списка "поднимаются" в памяти (сдвигаются по направлению возрастания адресов) подобно тому, как мыльные пузырьки поднимаются в воздух. При пузырьковой сортировке элементы списка просматриваются последовательно, начиная с первого, и каждый элемент списка сравнивается со следующим.

Если программа пузырьковой сортировки находит элемент, который превышает значение соседнего элемента с более старшим адресом, то она меняет эти элементы местами. Затем она сравнивает следующую пару соседних элементов, при необходимости меняет их местами и т. д. В момент, когда микропроцессор 8088 доходит до последнего элемента, в самый конец списка "всплывает" элемент с наибольшим значением.

При выполнении сортировки по этому методу микропроцессор обычно выполняет несколько проходов по списку. На рис. 5.2 показано, что после первого прохода элемент 50 "всплывает" в самый конец списка, а на следующих двух проходах "всплывают" на свои места элементы 40 и 30. Таким образом, этот список сортируется за три прохода.

Глядя на "моментальные снимки" списка, сделанные после каждого прохода, *Вам* нетрудно определить, в какой момент список становится полностью отсортированным, но как об этом узнает *ЭВМ*? Если Вы не установите счетчик числа повторений или не укажете каким-либо иным способом, что надо завершить сортировку, то ЭВМ будет бодро повторять проход за проходом до бесконечности. Так как число проходов сортировки зависит от начального порядка элементов в списке, то мы не можем заранее задать счетчик числа проходов. В качестве альтернативы мы можем установить специальный индикатор, называемый флагом обмена, который ЭВМ может использовать для определения момента прекращения сортировки.

Флаг обмена полагается равным 1 перед каждым проходом сортировки. Если при очередном проходе сортировки произошел обмен элементов, то значение флага изменяется на 0. Следовательно, ЭВМ по значению флага обмена после выполнения прохода может определить, надо ли ей продолжать сортировку: 0 означает, что надо выполнить еще один проход по списку; 1 означает, что список отсортирован и сортировку надо закончить. На рис. 5.3 показана блок-схема алгоритма пузырьковой сортировки.

Легко видеть, что даже при упорядоченном с самого начала списке микропроцессору придется выполнить один проход для установления факта упорядоченности. Если минимальное число проходов равно 1, то каково *максимальное* число проходов? Так как список из пяти элемен-

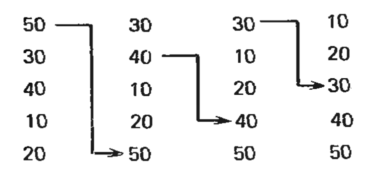


Рис. 5.2. "Всплывание" наибольших чисел в конец списка при пузырьковой сортировке

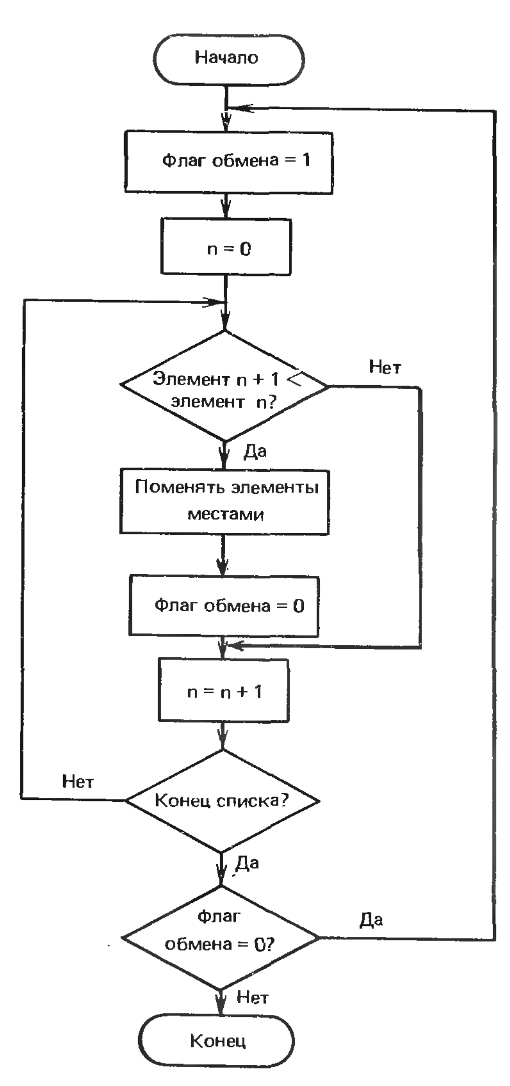


Рис. 5.3. Алгоритм пузырьковой сортировки

тов в примере 5.2 уже частично упорядочен, то для его сортировки в порядке возрастания потребовалось всего три прохода. Еще один проход нужен, чтобы удостовериться в упорядоченности списка, итого – четыре прохода.

Если бы этот список первоначально был упорядочен по убыванию (наихудший случай), то процессору пришлось бы выполнить пять проходов по списку: четыре для сортировки данных и один для выяснения того, что сортировка более не требуется. Отсюда мы можем заключить, что *для сортировки списка из N элементов требуется выполнить от одного до N проходов, а в среднем – (N + 1)/2 проходов.*

ПРОГРАММА ПУЗЫРЬКОВОЙ СОРТИРОВКИ

Изложенная выше теория пузырьковой сортировки и блок-схема подготовили нас к написанию программы сортировки списка. В примере 5.4 показана процедура сортировки списка слов B_SORT. (Ее легко модифицировать для сортировки списка байтов).

Как обычно, список находится в дополнительном сегменте, а его начальный адрес – в регистре DI. Процедура B_SORT использует регистр BX в качестве флага обмена, а регистр AX – для загрузки значения элемента, который сравнивается со следующим элементом списка.

Кроме того, процедура B_SORT использует две переменные, определенные в сегменте данных: SAVE_CNT, содержащую счетчик числа сравнений (равный числу элементов списка без единицы), и START_ADDR, содержащую начальный адрес списка. Процедура B_SORT использует их для восстановления исходных значений регистров CX и DI перед началом каждого нового прохода сортировки.

ПРИМЕР 5.4. ПРОЦЕДУРА ПУЗЫРЬКОВОЙ СОРТИРОВКИ

```
; Эта процедура упорядочивает методом пузырьковой сортировки
; 16-битовые элементы списка в памяти по убыванию их значений
; Список находится в дополнительном сегменте; его адрес берется
; из регистра DI. Длина списка (в словах) находится в его первой
; ячейке
; По возвращению из процедуры значение регистра DI не изменяется
;
; Следующие ниже переменные определите в сегменте данных
;
SAVE_CNT       DW   ?
START_ADDR     DW   ?
;
B_SORT    PROC
          PUSH   AX                ;Сохранить рабочие регистры
          PUSH   BX
          PUSH   CX
          MOV    START_ADDR,DI     ;Сохранить начальный адрес
          MOV    CX,ES:[DI]        ;Извлечь счетчик элементов
          DEC    CX                ;Подготовиться к (счетчик-1) сравнениям
          MOVE   SAVE_CNT,CX       ;Сохранить это значение в памяти
INIT:     MOV    BX,1              ;Положить флаг обмена (BX) = 1
          MOV    CX,SAVE_CNT       ;Загрузить счетчик в регистр CX,
          MOV    DI,START_ADDR     ; а начальный адрес - в регистр DI
NEXT:     ADD    DI,2              ;Адресоваться к элементу данных
          MOV    AX,ES:[DI]        ; и загрузить его в регистр AX
          CMP    ES:[DI+2],AX      ;Следующий элемент меньше текущего?
          JAE    CONT              ; Нет. Проверить следующую пару
          XCHG   ES:[DI+2],AX      ; Да. Поменять эти элементы местами
          MOV    ES:[DI],AX
          SUB    BX,BX             ;  и обнулить флаг обмена
```

```
CONT:       LOOP   NEXT            ;Обработать весь список
            CMP    BX,0            ;Были сделаны обмены значениями?
            JE     INIT            ; Да. Выполнить новый проход
            MOV    DI,START_ADDR   ; Нет. Восстановить регистры
            POP    CX
            POP    BX
            POP    AX
            RET                    ;  и выйти из процедуры
B_SORT      ENDP
```

Процедура B_SORT довольно простая, хотя в ней много команд. После вычисления значений переменных SAVE_CNT и START_ADDR процедура присваивает начальные значения флагу обмена (BX = 1), счетчику числа сравнений CX и указателю текущего элемента DI. Начиная с метки NEXT, микропроцессор 8088 загружает текущий элемент в регистр AX, а затем сравнивает его со следующим элементом.

Если второй элемент меньше первого, то микропроцессор 8088 загружает второй элемент в регистр AX и запоминает оба значения в памяти в обратном порядке. Так как тем самым осуществлен обмен элементов, микропроцессор обнуляет регистр BX. Команда LOOP с меткой CONT передает управление обратно к метке NEXT до тех пор, пока не будет обработан весь список.

Когда будут выполнены сравнения всех пар элементов, команда CMP проверит, равен ли 0 флаг обмена BX. Если равен, то на только что выполненном проходе был осуществлен по крайней мере один объем, и в этом случае микропроцессор 8088 переходит обратно к метке INIT, чтобы начать новый проход. В противном случае (если флаг BX остался равным 1 после выполнения прохода сортировки) список оказывается полностью отсортированным и микропроцессор 8088 восстанавливает регистр DI из переменной START_ADDR, а регистры BX и AX из стека и затем возвращает управление в основную программу.

ОПТИМИЗАЦИЯ ПРОГРАММЫ ПУЗЫРЬКОВОЙ СОРТИРОВКИ

Процедура пузырьковой сортировки из примера 5.4 имеет небольшой, но достаточно неприятный недостаток: она бесцельно сортирует некоторые элементы. А именно, на очередном проходе сортировки процедура B_SORT сравнивает каждую пару элементов списка, не считаясь с тем, что каждый проход заставляет один элемент списка "всплыть" на более высокое место. Иначе говоря, при первом проходе самый большой элемент "всплывает" в конец списка, при втором проходе второй по значению элемент "всплывает" на предпоследнее место и т. д. Следовательно, сдвинутые к концу списка элементы занимают окончательное (отсортированное) положение, поэтому их можно исключать из следующих проходов сортировки!

Для исключения отсортированных элементов *при каждом следующем проходе по списку надо выполнять на одно сравнение меньше, чем при предыдущем.* Этого можно добиться, если изменить нашу процедуру так, чтобы значение переменной SAVE_CNT уменьшалось на единицу перед каждым новым проходом.

Для этого надо так изменить шестую, седьмую и восьмую команды процедуры, чтобы операция уменьшения счетчика оказалась с меткой INIT. Кроме того, после нее надо добавить еще одну команду, заставляющую микропроцессор выйти из цикла, если переменная SAVE_CNT равна нулю.

Приведем перечень всех изменений:

Исходный вариант	Модификация
<pre> DEC CX MOV SAVE_CNT,CX INIT: MOV BX,1</pre>	<pre> MOV SAVE_CNT,CX INIT: MOV BX,1 DEC SAVE_CNT JZ SORTED</pre>

Здесь метка SORTED должна быть приписана команде MOV, восстанавливающей содержимое регистра DI из переменной START_ADDR. В примере 5.5 показана новая процедура сортировки (BUBBLE), полученная после внесения этих изменений.

Для любого заданного списка процедура BUBBLE выполняет то же число проходов сортировки, что и процедура B_SORT из примера 5.4. Но так как процедура BUBBLE осуществляет примерно вдвое меньше *сравнений*, то она выполняется гораздо быстрее, чем процедура B_SORT.

Например, при сортировке 100 элементов, расположенных в порядке убывания, процедура B_SORT осуществит 100 проходов, на каждом из которых будет выполнено 99 сравнений, т. е. всего 9900 сравнений. В отличие от нее процедура BUBBLE осуществит 99 проходов, выполняя 99 сравнений на первом проходе и одно на последнем, т. е. в среднем по 50 сравнений (всего 4950 сравнений).

Для сравнения процедур BUBBLE и B_SORT автор отсортировал с помощью каждой из них два списка 16-битовых элементов. Оба списка первоначально были упорядочены по убыванию. Первый из них, имевший 500 элементов, был отсортирован процедурой B_SORT за 7,5 с, а процедурой BUBBLE – за 4,5 с. Второй список, имевший 1000 элементов, был отсортирован за 28,0 с процедурой B_SORT и за 16,5 с процедурой BUBBLE. На основании этих результатов можно заключить, что процедура *BUBBLE сортирует список примерно на 40% быстрее, чем процедура B_SORT*.

Учтите, что время сортировки катастрофически возрастает с удлинением списков. Представленная здесь процедура пузырьковой сортировки может сортировать списки длиной до 32К слов, но при столь длинном списке Вам придется долго ждать. Действительно, в наихудшем случае список всего из 2000 слов сортируется процедурой BUBBLE за 66 с.

ПРИМЕР 5.5. УЛУЧШЕННАЯ ПРОЦЕДУРА ПУЗЫРЬКОВОЙ СОРТИРОВКИ

```
; Эта процедура упорядочивает методом пузырьковой сортировки
; 16-битовые элементы списка в памяти по убыванию их значений
; Список находится в дополнительном сегменте; его адрес берется
; из регистра DI. Длина списка (в словах) находится в его первой
; ячейке
; По возвращению из процедуры значение регистра DI не изменяется
;
; Следующие ниже переменные определите в сегменте данных
;
SAVE_CNT      DW   ?
START_ADDR    DW   ?
;
BUBBLE     PROC
           PUSH  AX               ;Сохранить рабочие регистры
           PUSH  BX
           PUSH  CX
           MOV   START_ADDR,DI    ;Сохранить начальный адрес
           MOV   CX,ES:[DI]       ;Извлечь счетчик элементов
           DEC   CX               ;Подготовиться к (счетчик-1) сравнениям
           MOVE  SAVE_CNT,CX      ;Сохранить это значение в памяти
```

```
INIT:     MOV   BX,1             ;Положить флаг обмена (BX) = 1
          DEC   SAVE_CNT         ;Подготовиться к (счетчик-1) сравнениям
          JZ    SORTED           ;Выйти, если SAVE_CNT = 0
          MOV   CX,SAVE_CNT      ;Загрузить число сравнений в регистр CX,
          MOV   DI,START_ADDR    ; а начальный адрес - в регистр DI
NEXT:     ADD   DI,2             ;Адресоваться к элементу данных
          MOV   AX,ES:[DI]       ; и загрузить его в регистр AX
          CMP   ES:[DI+2],AX     ;Следующий элемент меньше текущего?
          JAE   CONT             ; Нет. Проверить следующую пару
          XCHG  ES:[DI+2],AX     ; Да. Поменять эти элементы местами
          MOV   ES:[DI],AX
          SUB   BX,BX            ;  и обнулить флаг обмена
CONT:     LOOP  NEXT             ;Обработать весь список
          CMP   BX,0             ;Были сделаны обмены значениями?
          JE    INIT             ; Да. Выполнить новый проход
SORTED:   MOV   DI,START_ADDR    ; Нет. Восстановить регистры
          POP   CX
          POP   BX
          POP   AX
          RET                    ;  и выйти из процедуры
BUBBLE    ENDP
```

5.3. УПОРЯДОЧЕННЫЕ СПИСКИ

Теперь, когда Вы научились сортировать списки, обсудим поиск заданного значения, а также вставку и удаление элементов.

ПОИСК В УПОРЯДОЧЕННОМ СПИСКЕ

В разд. 5.1 мы усвоили, что поиск значения в неупорядоченном списке требует последовательного просмотра его элементов. Если список состоит из N элементов, этот процесс требует выполнения в среднем N/2 сравнений. Но если список *упорядочен*, то для поиска значения можно воспользоваться целым рядом методов. Большинство из них быстрее и эффективнее последовательного поиска для всех списков, кроме самых коротких.

БИНАРНЫЙ ПОИСК

Одним из распространенных методов поиска в упорядоченных списках является *бинарный поиск*. Это название отражает тот факт, что при его применении список последовательно делится на убывающие половины ("би" на латинском языке означает "два"), которые постепенно смыкаются на одном элементе. По этому методу поиск начинается с середины списка. Сначала определяется, где находится искомое значение, выше или ниже этой точки. Затем выбирается *соответствующая* половина списка, снова делится пополам и т. д.

Изображенная на рис. 5.4 блок-схема показывает, как выполнить бинарный поиск в упорядоченном списке и получить в качестве результата адрес элемента. Если искомое значение присутствует в списке, то адрес будет указывать на соответствующий элемент. Если такого значения в списке нет, то адрес укажет на последнюю ячейку, с которой было проведено сравнение. Конечно, выполняющая такой поиск программа должна каким-то образом возвращать признак того, отражает этот адрес успешный или безуспешный поиск.

Приведенную в примере 5.6 процедуру B_SEARCH можно использовать для поиска в упорядоченном списке слов без знака. Тот факт, что эта процедура опе-

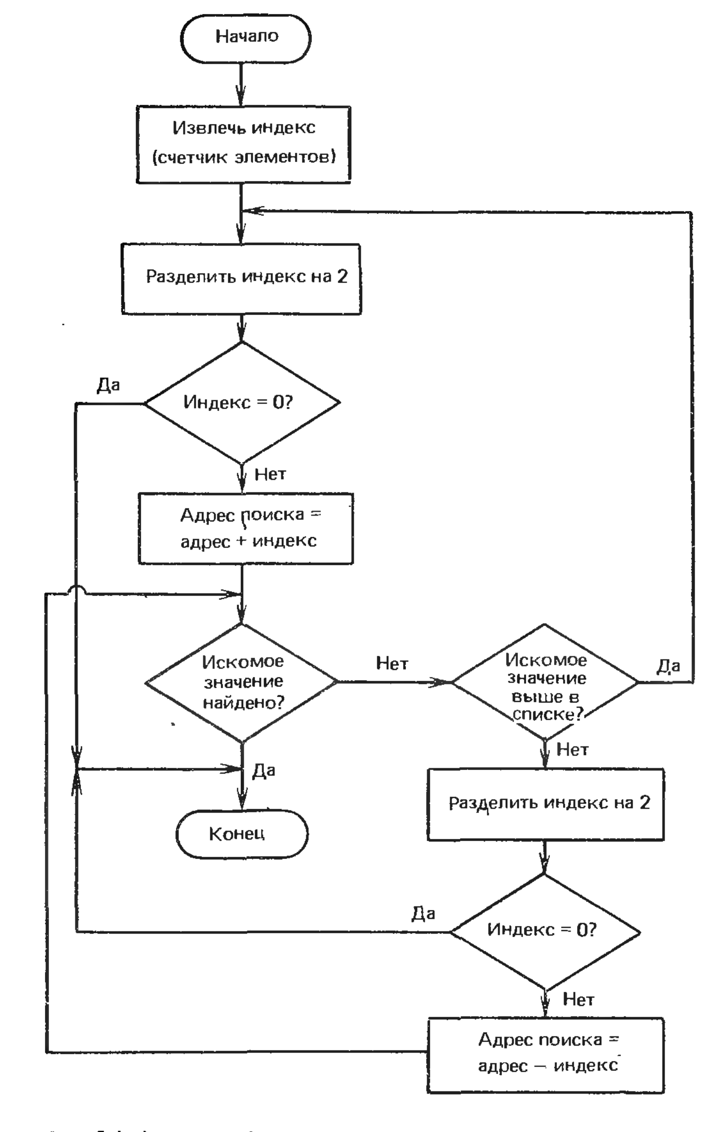


Рис. 5.4. Алгоритм бинарного поиска

рирует словами, а не байтами, заставляет нас внести несколько изменений в наш основной алгоритм.

Поскольку слова памяти отстоят друг от друга на два байта, мы должны включить команды, обеспечивающие четные значения индекса. По той же причине мы заканчиваем поиск и объявляем его безуспешным, если индекс уменьшился до 2 (вместо 0).

В этой процедуре искомое значение извлекается из регистра AX, а начальный адрес списка – из регистра DI. Процедура B_SEARCH возвращает адрес искомого значения в регистре SI, а флаг CF показывает, найдено оно (CF = 0) или нет (CF = 1).

ПРИМЕР 5.6. ПРОЦЕДУРА БИНАРНОГО ПОИСКА В СПИСКЕ 16-БИТОВЫХ ЭЛЕМЕНТОВ

```
; Эта процедура выполняет поиск заданного значения длиной в слово,
; содержащегося в регистре AX, в упорядоченном списке, находящемся
; в дополнительном сегменте
; Начальный адрес списка берется из регистра DI. Длина списка в
; словах находится в первой ячейке списка
; Результаты возвращаются через регистр SI и флаг переноса CF по
; следующим правилам:
;    1. Если значение найдено в списке, то флаг CF равен 0 и регистр
;       SI содержит адрес совпадающего с ним элемента
;    2. Если значение в списке не найдено, то флаг CF равен 1 и ре-
;       гистр SI содержит адрес последнего элемента, с которым было
;       проведено сравнение
; В любом случае значения регистров AX и DI не изменяются
;
; Следующую переменную определите в памяти:
;
START_ADDR   DW   ?
;
B_SEARCH     PROC
;
; Сначала определить, не выходит ли значение регистра AX за границы
; списка
;
             CMP   AX,ES:[DI+2]    ;Искомое значение < или = первому элементу?
             JA    CHK_LAST        ; Нет. Сравнить с последним элементом
             LEA   SI,ES:[DI+2]    ; Да. Извлечь адрес первого элемента
             JE    EXIT_1ST        ;Если значение = 1-й элемент, выйти
             STC                   ;Если значение < 1-й элемент,
;                                    установить флаг CF
EXIT_1ST:    RET
CHK_LAST:    MOV   SI,ES:[DI]      ;Указать на последний элемент
             SHL   SI,1
             ADD   SI,DI
             CMP   AX,ES:[SI]      ;Искомое значение > или = последнему эл.?
             JB    SEARCH          ; Нет. Искать в списке
             JE    EXIT_LAST       ; Да. Выйти, если значение = последний
;                                    элемент
             STC                   ;Если значение > последний элемент,
;                                    установить флаг CF
EXIT_LAST:   RET
;
; Искать значение в списке
;
SEARCH:      MOV   START_ADDR,DI   ;Сохранить адрес начала списка в памяти
             MOV   SI,ES:[DI]      ;Извлечь индекс
EVEN_IDX:    TEST  SI,1            ;Принудительно сделать его четным
             JZ    ADD_IDX
             INC   SI
ADD_IDX:     ADD   DI,SI           ;Вычислить адрес следующего элемента
COMPARE:     CMP   AX,ES:[DI]      ;Искомое значение найдено?
             JE    ALL_DONE        ; Если да, то выйти из процедуры
             JA    HIGHER          ; В противном случае найти нужную
;                                     половину списка
;
; Искать в левой части списка
;
             CMP   SI,2            ;Индекс = 2?
             JNE   IDX_OK
NO_MATCH:    STC                   ; Если да, то установить флаг CF
             JE    ALL_DONE        ;  и выйти из процедуры
IDX_OK:      SHR   SI,1            ; Если нет, разделить индекс на 2
             TEST  SI,1            ;Принудительно сделать его четным
             JZ    SUB_IDX
             INC   SI
```

```
SUB_IDX:      SUB   DI,SI              ;Вычислить адрес следующего элемента
              JMP   SHORT COMPARE      ;Проверить этот элемент
;
; Искать в правой части списка
;
HIGHER:       CMP   SI,2               ;Индекс = 2?
              JE    NO_MATCH           ; Если да, то установить флаг CF и
;                                        выйти из процедуры
              SHR   SI,1               ; Если нет, разделить индекс на 2
              JMP   SHORT EVEN_IDX     ; и перейти к проверке следующего
;                                        элемента
;
; Команды выхода из процедуры
;
ALL_DONE:     MOV   SI,DI              ;Занести адрес последнего сравнения
;                                        в регистр SI
              MOV   DI,START_ADDR      ;Восстановить адрес начала списка
              RET                      ; и выйти из процедуры
B SEARCH:     ENDP
```

Начальный шаг процедуры B_SEARCH не входит в основной алгоритм бинарного поиска: он сравнивает искомое значение с первым и последним элементами списка. Если искомое значение меньше первого или больше последнего элемента, или совпадает с одним из них, то процедура сразу же завершается. Если эти начальные проверки не срабатывают, то микропроцессор 8088 переходит к выполнению операции поиска, начинающейся с метки SEARCH.

Сначала микропроцессор 8088 сохраняет значение регистра DI в памяти, а затем копирует индекс (счетчик слов) из первой ячейки списка в регистр SI и дополняет его до ближайшего четного значения. Этот индекс добавляется к значению регистра DI, в результате чего получается адрес среднего элемента списка – отправная точка бинарного поиска. Затем микропроцессор 8088 определяет, продолжать ему поиск в верхней половине списка (выполняя команды с метки HIGHER) или в нижней половине.

В обоих случаях есть общие операции:

1. Проверить, не равен ли индекс 2. Если равен, то микропроцессор 8088 полагает флаг CF равным 1 (что означает безуспешный поиск) и передает управление метке ALL_DONE для выхода из процедуры.
2. Разделить индекс на 2 с помощью сдвига его вправо на один бит.
3. Дополнить новое значение индекса до ближайшего четного числа.

Однако при поиске в нижней половине списка микропроцессор 8088 вычитает индекс (SI) из текущего адреса (DI), а в случае поиска в верхней половине он добавляет индекс к текущему адресу.

Этот процесс повторяется до тех пор, пока либо индекс не уменьшится до 2, либо не будет найдено искомое значение. В любом случае управление передается метке ALL_DONE, после чего содержимое регистра DI загружается в регистр SI, а затем из памяти извлекается начальное значение регистра DI.

Насколько бинарный поиск эффективнее последовательного просмотра элементов списка (такого, как в примерах 5.1 и 5.2)? В статье An Introduction to Algorithm Design (Computer, February 1979, pp. 66 – 78) Джон Л. Бентли констатирует, что при последовательном поиске в списке из N элементов выполняется в среднем N/2 сравнений, а при бинарном поиске – $\log_2 N$ сравнений. Следовательно, при последовательном поиске в списке из 100 элементов в среднем потребуется 50 сравнений, а при бинарном поиске – примерно семь!

Обычно поиск значения в списке выполняется для того, чтобы затем осуществить какие-либо операции. Типичными операциями являются вставка элемента в список или удаление его из списка. Посмотрим, как они выполняются.

ВСТАВКА ЭЛЕМЕНТА В УПОРЯДОЧЕННЫЙ СПИСОК

Для вставки элемента в упорядоченный список надо выполнить следующие четыре шага:

1. Найти в памяти место вставки элемента.
2. Освободить место для вставки с помощью сдвига всех бо́льших элементов на одну позицию вверх.
3. Поместить вставляемый элемент в только что освобожденную позицию.
4. Добавить единицу к счетчику элементов списка, регистрируя увеличение длины списка.

Только что разработанная нами процедура B_SEARCH полезна для определения места вставки элемента, так как она возвращает адрес элемента, на котором был прекращен поиск. Для завершения шага 1 необходимо установить, где вставлять новый элемент: *до* или *после* конечного элемента поиска. Это можно сделать с помощью сравнения вставляемого значения с конечным элементом поиска.

В примере 5.7 показана процедура ADD_TO_OL, которая выполняет четыре указанных выше шага. Она начинается с вызова процедуры B_SEARCH для выяснения, не находится ли вставляемое значение уже в списке. Напомним, что процедура B_SEARCH возвращает адрес в регистре SI, а в качестве индикатора найдено/не найдено использует флаг переноса CF.

ПРИМЕР 5.7. ВСТАВКА ЭЛЕМЕНТА В УПОРЯДОЧЕННЫЙ СПИСОК

```
; Эта процедура добавляет значение, содержащееся в регистре AX,
; к упорядоченному списку, находящемуся в дополнительном сегменте,
; при условии, что в списке такого значения нет. Начальный адрес
; списка берется из регистра DI. Длина списка в словах находится
; в его первой ячейке
; Значения регистров AX и DI не изменяются
; Для проведения поиска в списке используется процедура B_SEARCH
; из примера 5.6
;
EXTRN       B_SEARCH
ADD_TO_OL   PROC
            PUSH  SI
            PUSH  CX
            PUSH  BX
            CALL  B_SEARCH         ;Значение уже содержится в списке?
            JNC   GOODBYE          ; Да. Выйти из процедуры
            MOV   BX,SI            ; Нет. Скопировать адрес последнего
                                   ;   сравнения в регистр BX
            MOV   CX,ES:[DI]       ;Найти адрес последнего элемента
            SHL   CX,1
            ADD   CX,DI            ; и поместить его в регистр CX
            PUSH  CX               ;Сохранить конечный адрес в стеке
            SUB   CX,SI            ;Определить число перемещаемых слов
            SHR   CX,1
            CMP   AX,ES:[SI]       ;Надо ли перемещать последний
                                   ;  сравненный элемент
            JA    EXCLUDE
            INC   CX               ; Да. Увеличить счетчик перемещений
                                   ;   на 1
            JNZ   CHECK_CNT
```

```
EXCLUDE:    ADD    BX,2             ; Нет. Подстроить указатель места
;                                       вставки
CHECK_CNT:  CMP    CX,0             ;Счетчик перемещений = 0?
            JNE    MOVE_ELS
            POP    SI               ; Если да, поместить значение в конец
;                                     списка,
            MOV    ES:[SI+2],AX
            JMP    SHORT INC_CNT    ;  а затем перейти к увеличению
;                                        счетчика элементов
MOVE_ELS:   POP    SI               ;Начать перемещение от начала списка
            PUSH   BX               ;Сохранить адрес вставки в стеке
MOVE_ONE:   MOV    BX,ES:[SI]       ;Переместить один элемент списка влево
            MOV    ES:[SI+2],BX
            SUB    SI,2             ;Указать на следующий элемент
            LOOP   MOVE_ONE         ;Повторять, пока все элементы не будут
;                                     перемещены
            POP    BX               ;Извлечь адрес вставки
            MOV    ES:[BX],AX       ;Вставить значение регистра AX в список
INC_CNT:    INC    WORD PTR ES:[DI] ;Добавить 1 к счетчику элементов
GOODBYE:    POP    BX               ;Восстановить значения регистров
            POP    CX
            POP    SI
            RET                     ; и выйти из процедуры
ADD_TO_OL   ENDP
```

После возвращения управления из процедуры B_SEARCH процедура ADD_TO_OL проверяет состояние флага CF и немедленно завершает работу, если флаг CF равен 0 (поскольку это означает, что значение уже находится в списке). Но если флаг CF равен 1, то процедура сохраняет в регистре BX адрес, на котором был прекращен поиск, и вычисляет адрес последнего элемента (в регистре CX). Вычитая из него содержимое регистра SI, получаем число байтов памяти, которые должны быть передвинуты вверх, чтобы освободить место для вставляемого значения. Сдвиг этого результата вправо (т. е. деление на 2) дает число передвигаемых *слов*. Если вставляемое значение меньше того значения, с которым было сделано последнее сравнение, то и его надо передвинуть (для чего счетчик сдвигаемых слов CX увеличивается на единицу).

Дойдя до метки CHECK_CNT, микропроцессор 8088 проверяет счетчик сдвигаемых слов. Если он равен нулю, то вставляемый элемент должен быть просто добавлен к концу списка. В противном случае этот элемент должен быть вставлен в список, для чего требуется передвинуть все следующие за ним элементы на одно слово вверх.

Команды после метки MOVE_ELS передвигают элементы вверх один за другим, начиная с последнего элемента списка. Когда все они будут передвинуты, микропроцессор 8088 вставит элемент (содержимое регистра AX) в образовавшийся просвет, а затем увеличит счетчик элементов списка на единицу.

УДАЛЕНИЕ ЭЛЕМЕНТА ИЗ УПОРЯДОЧЕННОГО СПИСКА

Гораздо легче удалить элемент из упорядоченного списка, чем вставить его. Требуется лишь найти его в списке, передвинуть все следующие за ним элементы на одну позицию вниз и уменьшить счетчик элементов списка на единицу.

В примере 5.8 показана типичная процедура удаления элемента (DEL_OL), которая использует процедуру B_SEARCH из примера 5.6 для поиска удаляемой

"жертвы". Как обычно, начальный адрес списка находится в регистре DI, а значение удаляемого элемента – в регистре AX.

Если процедура B_SEARCH обнаруживает заданное значение в списке, то процедура DEL_OL использует возвращенные ею адрес и адрес последнего элемента списка для определения числа слов, которые надо передвинуть в памяти вниз. Это передвижение выполняется циклом из четырех команд, начинающимся с метки MOVEM. После того как микропроцессор 8088 передвинет последнее слово, он уменьшит счетчик элементов списка, зафиксировав удаление.

ПРИМЕР 5.8. УДАЛЕНИЕ ЭЛЕМЕНТА ИЗ УПОРЯДОЧЕННОГО СПИСКА

```
; Эта процедура удаляет значение, содержащееся в регистре AX,
; из упорядоченного списка, находящегося в дополнительном сегменте,
; при условии, что такое значение в списке есть. Начальный адрес
; списка берется из регистра DI. Длина списка в словах находится
; в его первой ячейке
; Значения регистров AX и DI не изменяются
; Для проведения поиска в списке используется процедура B_SEARCH
; из примера 5.6
;
EXTRN       B_SEARCH
DEL_OL      PROC
            PUSH  SI
            PUSH  CX
            PUSH  BX
            CALL  B_SEARCH        ;Значение уже содержится в списке?
            JC    ADIOS           ; Нет. Выйти из процедуры
            MOV   CX,ES:[DI]      ; Да. Найти адрес последнего элемента
            SHL   CX,1
            ADD   CX,DI           ;  и поместить его в регистр CX
            CMP   CX,SI           ;Удалению подлежит последний элемент?
            JE    CNT_M1          ; Да. Уменьшить счетчик элементов
            SUB   CX,SI           ; Нет. Вычислить количество
            SHR   CX,1            ;  перемещаемых элементов
MOVEM:      MOV   BX,ES:[SI+2]    ;Переместить один элемент списка вправо
            MOV   ES:[SI],BX
            ADD   SI,2            ;Указать на следующий элемент
            LOOP  MOVEM           ;Повторять, пока все элементы не будут
                                  ; перемещены
;
CNT_M1:     DEC   WORD PTR ES:[DI]  ;Уменьшить счетчик элементов на 1
ADIOS:      POP   BX              ;Восстановить значения регистров
            POP   CX
            POP   SI
            RET                   ; и выйти из процедуры
DEL_OL      ENDP
```

5.4. ТАБЛИЧНЫЕ ФУНКЦИИ

Многие программы хранят обрабатываемые значения в таблицах. Иногда такие таблицы содержат числа, на вычисление которых уходит много времени, например значения синуса ряда углов. В других случаях они содержат параметры, которые определенным образом связаны с вводимыми данными, но не могут быть по ним вычислены. Например, Вы можете ввести в ЭВМ чью-нибудь фамилию, чтобы получить в ответ номер телефона данного абонента.

В подобных приложениях используются так называемые *табличные функции*. Как можно заключить из их названия, табличные функции возращают элемент информации (функцию).

Табличными функциями можно заменить сложные или длительные преобразования, например извлечение квадратного или кубического корня либо вычисле-

ние тригонометрических функций (синуса, косинуса и т. д.). Табличные функции особенно эффективны, если они должны быть определены для небольшого числа значений аргумента (например, кубы чисел от 1 до 20). При применении табличной функции ЭВМ не требуется выполнять сложные вычисления всякий раз, когда необходимо получить значение функции по заданному значению аргумента.

Вообще говоря, табличные функции позволяют сэкономить время во всех случаях, кроме самых простейших. (Например, нет смысла использовать табличную функцию для хранения значений, которые получаются в результате удвоения аргумента.) Так как обычно табличные функции занимают большие объемы памяти, то их применение целесообразно в тех приложениях, где можно пожертвовать памятью ради ускорения исполнения программы.

Поскольку табличные функции очень распространены, то для работы с ними у микропроцессора 8088 есть специальная команда XLAT (translate – переводить с одного языка на другой). Она извлекает значение из таблицы байтов, используя содержимое регистра BX в качестве базового адреса и содержимое регистра AL в качестве индекса. Результат возвращается командой XLAT в регистре AL. В настоящем разделе приводятся примеры применения табличных функций, имеющих значения и байтов, и слов.

ТАБЛИЧНЫЕ ФУНКЦИИ В КАЧЕСТВЕ ЗАМЕНЫ ФОРМУЛ

Можно сократить время исполнения и разработки программы, если представить результаты вычисления сложных формул в виде табличной функции. В качестве типичного примера рассмотрим применение табличных функций для получения значений синуса или косинуса угла.

СИНУС УГЛА

Из курса тригонометрии Вам должно быть известно, что график синусов углов от 0 до 360° представляет собой кривую, изображенную на рис. 5.5, *а*. Приближенные значения синуса угла X[1] можно получить из формулы

$$\sin(X) = X - \frac{X^3}{3!} + \frac{X^5}{5!} - \frac{X^7}{7!} + \frac{X^9}{9!} - \ldots$$

Можно написать программу, выполняющую эти вычисления, но она будет исполняться в течение нескольких миллисекунд. Если в Ваших приложениях требуется, чтобы значения синусов имели высокую точность, то придется написать программу, но в большинстве приложений можно воспользоваться таблицей значений синусов углов.

Если в решаемой Вами задаче требуется иметь значения синуса любого угла от 0 до 360°, измеряемого целым числом градусов, то сколько значений синусов должна содержать таблица? 360? Нет, мы можем ограничиться таблицей, содержащей только 91 значение синусов по одному для каждого угла от 0 до 90°.

Чтобы понять, как это можно сделать, посмотрим на рис. 5.5 еще раз. Назовем левую четверть графика (углы от 0 до 90°) первым квадрантом. Тогда:

1. Синусы углов второго квадранта (от 91 до 180°) являются "зеркальным отражением" синусов углов первого квадранта.

[1] Измеряемого в радианах. – *Прим. перев.*

2. Синусы углов третьего квадранта (от 181 до 270°) являются "негативным обращением" синусов углов первого квадранта.

3. Синусы углов четвертого квадранта (от 271 до 360°) являются "негативным обращением и зеркальным отражением" синусов углов первого квадранта.

Таким образом, синусы углов второго – четвертого квадрантов очень просто связаны с синусами углов первого квадранта.

Следовательно, если организовать значения синусов углов первого квадранта в виде табличной функции, то Ваша программа может найти синус угла в любом квадранте, осуществив следующие преобразования:

Положение угла	Преобразования
0° <= X <= 90°	sin (X)
91° <= X <= 180°	sin (180° – X)
181° <= X <= 270°	– sin (X – 180°)
271° <= X <= 360°	– sin (360° – X)

По этим соотношениям можно составить блок-схему программы преобразования значения угла в его синус (рис. 5.6). По этой блок-схеме значение синуса возвращается в виде пары знак-амплитуда: старший бит результата дает знак синуса (0 – положительный, 1 – отрицательный), а остальные биты содержат амплитуду, т. е. абсолютное значение синуса.

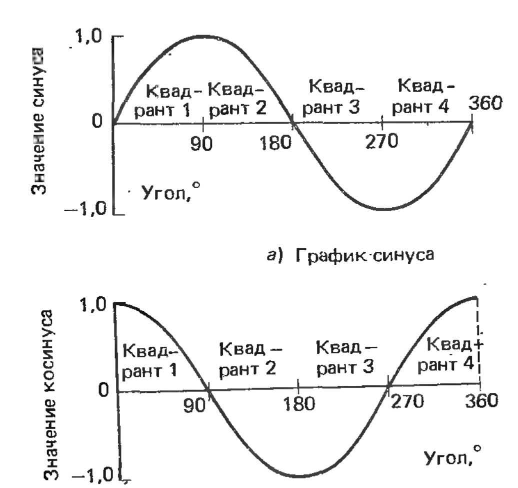


Рис. 5.5. Синусы и косинусы углов от 0 до 360°

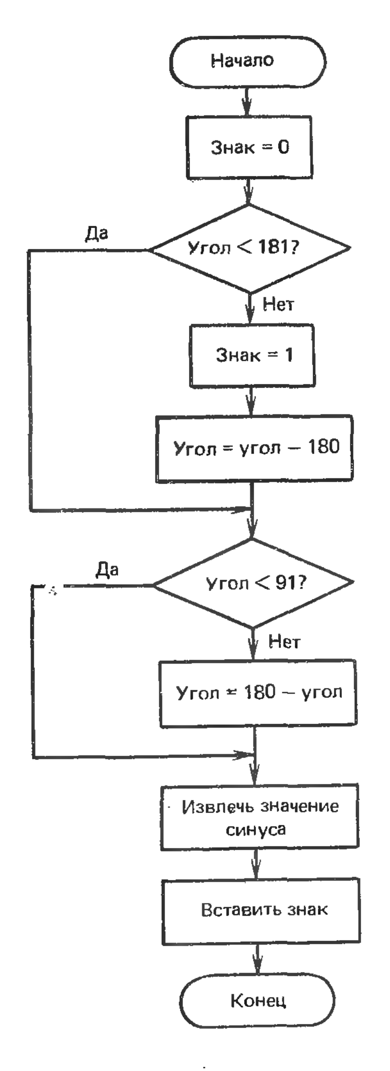


Рис. 5.6. Блок-схема программы преобразования синуса угла

В примере 5.9 показана процедура FIND_SINE, в которой табличная функция используется для преобразования угла в его синус. Она получает значение угла от 0 до 360° в регистре AX и возвращает 16-битовое значение синуса (в форме знак-амплитуда) в регистре BX. Амплитуды синусов хранятся в виде целых чисел в таблице SINES. Они должны быть разделены на 10 000 перед использованием их в качестве операндов.

Сначала микропроцессор 8088 проверяет величину угла. Если он меньше 181°, то микропроцессор переходит к метке SIN_POS. В противном случае он полагает старший бит регистра CX равным 1 (этот регистр используется для хранения знака), поскольку синусы углов больше 180° отрицательны, а затем вычитает 180° из значения угла.

ПРИМЕР 5.9. ПРИМЕНЕНИЕ ТАБЛИЧНОЙ ФУНКЦИИ ДЛЯ ПОЛУЧЕНИЯ СИНУСА УГЛА

```
; Эта процедура возвращает значение синуса угла (от 0 до 360
; градусов), содержащегося в регистре AX
; Синус угла, в форме "знак-амплитуда", возвращается в регистре BX
; Содержимое регистра AX не изменяется
;
; Определите эту таблицу значений синуса в сегменте данных
;
SINES   DW   0,175,349,523,698,872          ;0-5
        DW   1045,1219,1392,1564,1736       ;6-10
        DW   1908,2079,2250,2419,2588       ;11-15
        DW   2756,2924,3090,3256,3420       ;16-20
        DW   3584,3746,3907,4067,4226       ;21-25
        DW   4384,4540,4695,4848,5000       ;26-30
        DW   5150,5299,5446,5592,5736       ;31-35
        DW   5878,6018,6157,6293,6428       ;36-40
        DW   6561,6691,6820,6947,7071       ;41-45
        DW   7193,7313,7431,7547,7660       ;46-50
        DW   7771,7880,7986,8090,8191       ;51-55
        DW   8290,8387,8480,8572,8660       ;56-60
        DW   8746,8829,8910,8988,9063       ;61-65
        DW   9135,9205,9272,9336,9397       ;66-70
        DW   9455,9511,9563,9613,9659       ;71-75
        DW   9703,9744,9781,9816,9848       ;76-80
        DW   9877,9903,9926,9945,9962       ;81-85
        DW   9976,9986,9994,9998,10000      ;86-90
;
; Далее следует процедура работы с табличной функцией
;
FIND_SINE  PROC
           PUSH  AX          ;Сохранить значения регистров AX
           PUSH  CX          ; и CX
           SUB   CX,CX       ;Положить знак равным нулю
           CMP   AX,181      ;Угол < 181?
           JB    SIN_POS     ; Да. Продолжить со знаком = 0
           MOV   CX,8000H    ; Нет. Положить знак = 1
           SUB   AX,180      ;  и вычесть 180 из угла
SIN_POS:   CMP   AX,91       ;Угол < 91?
           JB    GET_SIN     ; Да. Извлечь из таблицы значение синуса
           NEG   AX          ; Нет. Вычесть угол из 180
           ADD   AX,180
GET_SIN:   MOV   BX,AX       ;Сделать угол индексом слова
           SHL   BX,1
           MOVE  BX,SINE[BX] ; и извлечь значение синуса
           OR    BX,CX       ;Скомбинировать синус и бит знака
           POP   CX
           POP   AX
           RET
FIND_SINE  ENDP
```

После того как знак результата помещен в регистр CX, команда CMP с меткой SIN_POS сравнивает значение заданного угла с 91°. Если угол больше или равен 91°, то его надо вычесть из 180°.

Эту операцию может выполнить команда SUB 180, AX, но микропроцессор 8088 не допускает такую форму команды. Это как раз тот случай, когда мы выполняем вычитание с помощью обращения знака регистра AX и добавления 180 к полученному результату. Четыре команды, указанные после метки GET_SIN, загружают значение угла в регистр BX, удваивают его для получения индекса слова, извлекают значение синуса из таблицы SINES, а затем с помощью операции OR добавляют к нему знак из регистра CX.

Время, затрачиваемое процедурой FIND_SINE на получение значения синуса угла, зависит от того, какому квадранту принадлежит этот угол. Время исполнения (исключая время исполнения команд CALL и RET) этой процедуры зависит от значения угла:

для углов от 0 до 90° процедура FIND_SINE исполняется
за 125 тактов, или 26,25 мкс;
для углов от 91 до 180° процедура FIND_SINE исполняется
за 120 тактов, или 25,20 мкс;
для углов от 181 до 270° процедура FIND_SINE исполняется
за 121 такт, или 25,41 мкс;
для углов от 271 до 360° процедура FIND_SINE исполняется
за 116 тактов, или 24,36 мкс.

КОСИНУС УГЛА

Как показано на рис. 5.5, *б*, график косинуса представляет собой не что иное, как смещенный на один квадрант влево график синуса. Следовательно, косинус любого угла равен синусу угла, на 90° большего. Таким образом

$$\cos(X) = \sin(X + 90).$$

Благодаря этому тождеству мы можем использовать таблицу SINES из примера 5.9 для получения значений как синусов, так и косинусов заданного угла. В примере 5.10 показано получение косинуса угла. Как и в случае процедуры FIND_SINE, возвращаемый процедурой FIND_COS результат надо разделить на 10 000.

Учтите, что графики синуса и косинуса симметричны относительно начала координат и вертикальной оси, поэтому синусы и косинусы отрицательных углов имеют те же амплитуды, что и синусы и косинусы противоположных им положительных углов. Например, синус угла −25° совпадает по амплитуде с синусом угла + 25°, а косинус угла −25° – с косинусом угла + 25° Следовательно, процедурами FIND_SINE и FIND_COS можно пользоваться и для углов от −1 до −360°, передавая в регистре AX *абсолютное* значение угла[1].

---

[1] При этом знак синуса корректируется. – *Прим. перев.*

ПРИМЕР 5.10. ПРИМЕНЕНИЕ ТАБЛИЧНОЙ ФУНКЦИИ ДЛЯ ПОЛУЧЕНИЯ КОСИНУСА УГЛА

```
; Эта процедура возвращает значение косинуса угла (от 0 до 360
; градусов), содержащегося в регистре AX
; Косинус угла, в форме "знак-амплитуда", возвращается в регистре BX
; Содержимое регистра AX не изменяется
; Эта процедура вызывает процедуру FIND_SINE (пример 5.9)
;
EXTRN     FIND_SINE:FAR
FIND_COS  PROC
          PUSH  AX          ;Сохранить значение регистра AX
          ADD   AX,90       ;Добавить 90 для обращения к FIND_SINE
          CMP   AX,360      ;Результат превышает 360?
          JNA   GET_COS
          SUB   AX,360      ; Если да, то вычесть из него 360
GET_COS:  CALL  FIND_SINE   ;Извлечь табличное значение косинуса
          POP   AX
          RET
FIND_COS  ENDP
```

ТАБЛИЧНЫЕ ФУНКЦИИ И ПРЕОБРАЗОВАНИЕ КОДОВ

Таблицы могут содержать и закодированные данные, например, управляющие коды дисплея и принтера, а также сообщения. В примере 5.11 показана процедура, использующая многозначную табличную функцию. Она преобразует шестнадцатеричную цифру, переданную в регистре AL, в ее эквиваленты в кодах ASCII, BCD и EBCDIC, возвращаемые в регистрах CH, CL и AH соответственно.

ПРИМЕР 5.11. ПРЕОБРАЗОВАНИЕ ШЕСТНАДЦАТЕРИЧНОЙ ЦИФРЫ В КОДЫ ASCII, BCD И EBCDIC

```
; Эта процедура преобразует шестнадцатеричную цифру, содержащуюся
; в регистре AL, в ее ASCII-, BCD- и EBCDIC-эквиваленты. Преобра-
; зованные значения возвращаются в регистрах CH, CL и AH соответ-
; ственно
; Содержимое регистра AL не изменяется
;
; Определите эти таблицы значений в сегменте данных
;
ASCII   DB '0123456789ABCDEF
BCD     DB 0,1,2,3,4,5,6,7,8,9,10H,11H,12H,13H,14H,15H
EBCDIC  DB 0F0H,0F1H,0F2H,0F3H,0F4H,0F5H,0F6H,0F7H
        DB 0F8H,0F9H,0C1H,0C2H,0C3H,0C4H,0C5H,0C6H
;
; Ниже приводится процедура преобразования
;
CONV_HEX  PROC
          PUSH  BX          ;Сохранить значения регистров BX
          PUSH  DX          ; и DX
          MOV   DL,AL       ;Сохранить входное значение в DL
          LEA   BX,ASCII    ;Извлечь ASCII-код
          XLAT  ASCII
          MOV   CH,AL       ; и загрузить его в регистр CH
          MOV   AL,DL
          LEA   BX,DCD      ;Извлечь BCD-код
          XLAT  BCD
          MOV   CL,AL       ; и загрузить его в регистр CL
          MOV   AL,DL
          LEA   BX,EBCDIC   ;Извлечь EBCDIC-код
          XLAT  EBCDIC
          MOV   AH,AL       ; и загрузить его в регистр AH
          MOV   AL,DL       ;Восстановить значения регистров
          POP   DX
          POP   BX
          RET
CONV_HEX  ENDP
```

При пересылке между ЭВМ и принтером, дисплеем или каким-либо другим периферийным устройством системы данные должны быть преобразованы в коды ASCII (American Standart Code for Information Interchange – американский стандартный код для обмена информацией). Код EBCDIC (Extended Binary-Coded Decimal Interchange Code – расширенный двоично-десятичный код для обмена данными) представляет собой протокол обмена информацией в информационных и телекоммуникационных системах.

ТАБЛИЦЫ ПЕРЕХОДОВ

Некоторые таблицы вместо значений данных содержат *адреса*. Например, процедура обработки ошибок может использовать табличную функцию для получения начального адреса одного из нескольких возможных сообщений. Аналогично процедура обработки прерывания может использовать табличную функцию для вызова одной из нескольких программ обработки прерывания в зависимости от того, какого рода обслуживание запросило данное устройство. Другие процедуры могут использовать табличные функции для вызова одной из нескольких подпрограмм в зависимости от того, какой выбор из предложенного меню сделал пользователь. Во всех этих (и многих других) приложениях таблица содержит адреса и называется *таблицей переходов*.

Пример 5.12 иллюстрирует приложение таблицы переходов, которое могло бы появиться в обучающей программе. Процедура SEL_OPT передает управление одной из четырех подпрограмм (ADDITION, SUBSTRACTION, MULTIPLICATION или DIVISION) в зависимости от того, выбрал студент из меню 0, 1, 2 или 3.

ПРИМЕР 5.12. ПРОЦЕДУРА ВЫБОРА ИЗ МЕНЮ

```
; Эта процедура вызывает одну из четырех подпрограмм в зависимости
; от кода выбора пользователя, содержащегося в регистре AL
; Содержимое регистров AX и DI изменяется
;
; Следующая таблица смещений адресов меток должна находиться в
; сегменте данных:
;
CHOICE   DW   ADDITION,SUBSTRACTION,MULTIPLICATION,DIVISION
;
; Ниже приводится процедура выбора
;
SEL_OPT     PROC   FAR
            CMP    AL,3             ;Ошибочный выбор?
            JA     ERROR
            CBW                     ; Нет. Преобразовать код в слово,
            MOV    DI,AX            ;  загрузить результат в DI
            SHL    DI,1             ;  и преобразовать его в индекс
            JMP    CHOICE[DI]       ;Перейти к подпрограмме
ERROR:      ...
            ...
            RET
ADDITION:
            ...
            ...
            RET
SUBSTRACTION:
            ...
            ...
            RET
MULTIPLICATION:
            ...
            ...
            RET
```

```
DIVISION:
            ...
            ...
            RET
SEL_OPT     ENDP
```

Процедура SEL_OPT проверяет, правилен ли переданный ей код, и переходит к подпрограмме выдачи сообщения об ошибке, если он превышает 3. Правильный код используется процедурой SEL_OPT в качестве индекса при выполнении косвенного перехода к выбранной подпрограмме. Когда студент заканчивает работу с этой подпрограммой, то по команде RET управление возвращается программе, вызвавшей процедуру SEL_OPT, – той программе, что изобразила меню на экране.

5.5. ТЕКСТОВЫЕ ФАЙЛЫ

В предыдущих разделах мы имели дело со структурами данных, элементами которых были числа. Однако при обработке текстов и в других приложениях приходится обрабатывать нечисловую информацию – в первую очередь текстовые файлы.

Текстовые файлы представляют собой списки, элементами которых являются строки символов в кодах ASCII. Например, текстовый файл с информацией о персонале предприятия будет содержать по одному элементу, или *записи,* для каждого служащего. В свою очередь, каждая запись состоит из нескольких *полей,* в которых перечисляются фамилия служащего, его табельный номер, смена, тарифная ставка и т. д. Для работы с текстовыми файлами особенно удобны команды обработки строк.

Текстовые файлы можно обрабатывать теми же методами, что и числовые файлы. Но так как записи текстовых файлов содержат много символов, то оперирующие ими программы должны кое в чем отличаться от тех, которые оперируют простыми списками байтов или слов.

Например, поиск в текстовом файле обычно производится так, что искомое значение сравнивается только с *частью* записи (скажем, с полем, содержащим фамилию), а не со всей записью. Аналогично при пузырьковой сортировке текстового файла операция сравнения смежных записей проводится также по значениям одного поля, но операция обмена перемещает записи *целиком.*

В качестве простого примера операции над текстовым файлом рассмотрим программу сортировки списка фамилий и телефонных номеров. Предполагается, что первая ячейка списка содержит двухбайтовый счетчик записей.

Каждая запись списка имеет длину 42 байта и разделена на три поля: фамилию длиной 15 байтов, имя и инициал отчества длиной 15 байтов и номер телефона длиной 12 байтов. Предполагается, что все неиспользованные байты поля содержат коды пробела (ASCII). Следовательно, типичная запись списка будет иметь следующий вид:

```
DB 'Иванов', 9 DUP (' ')
DB 'Семен П.', 7 DUP (' ')
DB '251-32-67', 3 DUP (' ')
```

(Конечно, обычно данные вводятся в текстовые файлы с клавиатуры. Но пока предположим, что они уже находятся в памяти.)

В примере 5.13 показана процедура PHONE_NOS, которая сортирует пузырьковым методом список телефонов, находящийся в дополнительном сегменте. По конструкции она похожа на "улучшенную процедуру пузырьковой сортировки" из примера 5.5.

```
; Эта процедура сортирует в алфавитном порядке телефонный список.
; Список содержит слово, в котором находится счетчик числа записей,
; а за ним идут собственно записи
; Каждая запись имеет длину 42 байта и состоит из трех полей:
; 15-байтовой фамилии, 15-байтового имени/инициала отчества и 12-бай-
; тового номера телефона
; Список находится в дополнительном сегменте, а его начальный адрес
; (адрес слова со счетчиком) - в регистре DI
;
; Определите следующие переменные в сегменте данных:
;
FIRST_ENT    DW   ?
SAVE_CNT     DW   ?
;
; Ниже следует основная процедура
;
PHONE_NOS    PROC
             PUSH  AX                ;Сохранить значения рабочих регистров
             PUSH  BX
             PUSH  CX
             PUSH  BP
             PUSH  DI
             PUSH  SI
             PUSH  DS
             CLD                     ;Положить (DF) = 0 для движения вправо
             MOV   CX,ES:[DI]        ;Извлечь счетчик записей
             MOV   SAVE_CNT,CX       ; и сохранить его в памяти
             ADD   DI,2              ;Получить адрес первой записи
             MOV   FIRST_ENT,DI      ; и сохранить его в памяти
INIT:        MOV   BX,1              ;Флаг обмена (BX) = 1
             DEC   SAVE_CNT          ;Подготовиться к (счетчик-1) сравнениям
             JZ    SORTED            ;Выйти из процедуры, если SAVE_CNT = 0
             MOV   CX,SAVE_CNT       ;Загрузить счетчик сравнений в CX
             MOV   BP,FIRST_ENT      ; и адрес первой записи в BP
NEXT:        MOV   DI,BP             ;Загрузить в DI указатель на первую
;                                      запись,
             MOV   SI,BP             ; а в SI - на следующую
             ADD   SI,42
             PUSH  CX                ;Сохранить текущий счетчик сравнений
             MOV   CX,15             ;Сравнить 15-байтовые поля с фамилиями
                                     ;Следующая фамилия < текущей?
REPE         CMPS  ES:BYTE PTR[SI],ES:[DI]
             JAE   CONT              ; Нет. Сравнить следующую пару
             MOV   CX,42             ; Да. Поменять записи местами
SWAPEM:      MOV   AL,ES:[BP]
             XCHG  ES:[BP+42],AL
             INC   BP
             LOOP  SWAPEM
             SUB   BX,BX             ;Положить флаг обмена = 0
CONT:        POP   CX                ;Загрузить счетчик сравнений заново
             LOOP  NEXT              ; и сравнить следующую пару фамилий
             CMP   BX,0              ;Сделан ли хоть один обмен?
             JE    INIT              ; Если да, выполнить новый проход
SORTED:      POP   DS                ; Если нет, восстановить значения
             POP   SI                ;  регистров
             POP   DI
             POP   BP
             POP   CX
             POP   BX
             POP   AX
             RET                     ;  и выйти из процедуры
PHONE_NOS    ENDP
```

После сохранения используемых в программе регистров процедура PHONE_NOS считывает счетчик записей и адрес первой записи в две переменные – SAVE_CNT и FIRST_ENT. Команды между метками NEXT и SWAPEM подготавливают и исполняют операцию CMPS. Здесь регистр DI указывает на поле фамилии текущей записи, а регистр SI – на поле фамилии следующей записи; эти поля отстоят в памяти на 42 байта.

Цикл, начинающийся меткой SWAPEM, меняет местами две записи, если в этом возникает необходимость. Так как записи имеют длину 42 байта, то счетчику цикла CX присваивается начальное значение 42. Остальная часть процедуры похожа на ту, что приведена в примере 5.5.

Обратите внимание на то, что в процедуре PHONE_NOS не учтена возможность существования однофамильцев. Если таковые имеются, то процедура должна сравнить их имена и отсортировать записи однофамильцев в алфавитном порядке имен. Попробуйте соответствующим образом модифицировать пример 5.13.

УПРАЖНЕНИЯ

1. Рассмотренные в этой главе процедуры обработки списков не проверяют, пуст ли список (пустой список имеет счетчик, содержащий 0, но в нем нет ни одного элемента данных) перед выполнением операций добавления, удаления или поиска. Чтобы исправить эту ошибку, модифицируйте пример 5.1 так, чтобы содержимое регистра AX становилось первым элементом списка, если он пуст. С помощью этой модифицированной процедуры можно не только добавлять элементы к существующим спискам, но и создавать новые списки.

2. Напишите процедуру, которая ищет в упорядоченном списке элемент со значением, равным содержимому регистра AX, и после его обнаружения *заменяет* этот элемент на содержимое регистра BX.

ГЛАВА 6. ПОЛЬЗОВАНИЕ СИСТЕМНЫМИ РЕСУРСАМИ

Содержание предыдущих глав этой книги относилось в основном к микропроцессору 8088 и, следовательно, к любой ЭВМ, выполненной на его базе. В этой главе мы покажем, как пользоваться встроенными ресурсами конкретной ЭВМ – IBM PC или XT.

Под встроенными ресурсами мы понимаем программы, образующие главную исполнительную программу ЭВМ: ее базовую систему ввода-вывода BIOS (Basic I/O system). В функцию BIOS входит запоминание символов, набираемых на клавиатуре, изображение символов на экране и обмен данными между устройствами, присоединенными к Вашей ЭВМ: дисководами, принтером и т. д.

Короче говоря, без BIOS персональная ЭВМ PC представляет собой всего лишь набор металлических и пластмассовых деталей и электронных компонентов; BIOS вкладывает в нее интеллект и превращает в ЭВМ.

Сначала мы кратко обсудим распределение памяти в ЭВМ PC и XT, затем опишем доступные программы, встроенные в BIOS, а также другие программы, которые дисковая операционная система DOS помещает в память. Большинство из них являются программами обработки прерываний; это означает, что Ваша программа может вызвать их с помощью исполнения соответствующей команды INT.

6.1. ПАМЯТЬ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

На рис. 6.1 показано распределение адресного пространства микропроцессора 8088 объемом в 1 Мбайт в персональных ЭВМ PC и XT. На основной *системной плате* ЭВМ установлена память, позволяющая чтение и запись данных. Эта память сокращенно именуется ЗУПВ (запоминающее устройство с произволь-

Адрес	Размер и назначение
00000	64K Стандартная память
10000	192K Дополнительная память, помещаемая на системной плате
40000	384K Область дополнительной памяти 384K
A0000	16K Зарезервированная память
A4000	112K Видеобуфер графических образов/текста
C0000	32K Область расширения ПЗУ
C8000	8K ПЗУ жесткого диска
CA000	168K Область ПЗУ платы адаптера
F4000	8K Область ПЗУ пользователя
F6000	32K ПЗУ Кассетного Бейсика
FE000	8K ПЗУ BIOS

Адрес	Назначение
00000	Векторы прерываний BIOS (0H – 1FH)
000B0	Векторы прерываний DOS (20H – 3FH)
00100	Зарезервированные векторы прерываний (40H – 5FH)
00180	Векторы прерываний пользователя (60H – 7FH)
00200	Векторы прерываний Бейсика (80H – FFH)
00400	Область данных BIOS
00500	Область данных Бейсика и DOS
00600	62,5K Память, доступная пользователю для чтения и записи

Рис. 6.1. Распределение памяти вычислительной системы

ной выборкой) и имеет объем минимум 64 Кбайт, что соответствует адресам от 0 до 0FFFF. Как показано на "увеличенном снимке" справа, не вся эта память доступна пользователям. Младшие 1,5 Кбайт содержат векторы прерываний (адреса программ обработки прерываний) и несколько рабочих областей памяти, используемых системой BIOS, операционной системой DOS и Бейсиком.

На системной плате можно дополнительно разместить 192 Кбайт памяти (с адресами от 10 000 до 3FFFF). Таким образом, всего на ней помещается до 256 Кбайт памяти. На добавочных платах можно разместить еще 384 Кбайт памяти (с адресами от 40 000 и выше), что дает возможность расширить объем памяти персональной ЭВМ до его предела 640 Кбайт.

Следом за добавочной областью памяти располагается блок из 16К ячеек, которые фирма IBM зарезервировала для каких-то таинственных будущих модификаций. С адреса A4000 начинается блок памяти в 112 Кбайт, где в IBM PC хранятся прообразы графических и алфавитно-цифровых изображений, выдаваемых на экран дисплея.

Оставшаяся часть адресного пространства, начинающаяся с адреса C0000, выделена под адреса постоянных запоминающих устройств (ПЗУ). Первые 32 Кбайта этой части памяти называются "областью расширения ПЗУ", но только специалистам фирмы известно, для чего эту область планируется использовать. Следующие 8 Кбайт, начиная с адреса C8000, соответствуют ячейкам ПЗУ адаптера жесткого диска IBM PC XT. Следующие 8 Кбайт, начиная с адреса F4000, соответствуют ПЗУ, содержимое которого может быть запрограммировано пользователем. Это ПЗУ вставляется в свободную панель, расположенную на основной плате. Наконец, последние 40 Кбайт с самыми старшими адресами (от F6000 до FFFFF) соответствуют ПЗУ кассетного Бейсика и ПЗУ BIOS.

Если в Вашем распоряжении есть Макроассемблер фирмы IBM или какой-либо иной перемещающий Ассемблер, то Вам не надо думать о том, в какой части памяти ЭВМ запоминает Ваши программы и данные. Однако Вы *должны* знать соответствие между прерываниями и системными программами, чтобы использовать встроенные прерывания или добавить к ним свои собственные.

Как показано в правой части рис. 6.1, система BIOS, операционная система DOS и Бейсик используют прерывания с типами 0 – 1FH, 20H – 3FH и 80H – FFH соответственно. Фирма IBM зарезервировала типы 40H – 5FH для будущего употребления; тем самым для Ваших нужд осталось 32 прерывания с типами 60H – 7FH.

6.2. ПРЕРЫВАНИЯ СИСТЕМЫ BIOS

Прерывания системы BIOS дают возможность воспользоваться богатым набором встроенных в IBM PC функций. В табл. 6.1 показано соответствие между векторами прерываний системы BIOS и этими функциями. Для полноты сведений в этой таблице указаны начальные значения номера блока и смещения адреса, которые присваиваются этим векторам в момент включения питания. Номер блока и смещение адреса разделены двоеточием.

Не пытайтесь использовать эти адреса в своей программе: они зависят от модели IBM PC и версии операционной системы DOS и приведены только для того, чтобы Вам было легче отыскать листинги программ обработки прерываний в техническом руководстве по IBM PC. В приложении A этого руководства содержится полный листинг системы BIOS, в левом столбце которого указаны смещения адресов (как показано на рис. 6.1, начальный адрес системы BIOS имеет номер блока F000). Например, чтобы найти программу PRINT_SCREEN (распечатать содержимое экрана), обрабатывающую прерывание типа 5, надо просматривать листинг до тех пор, пока Вы не дойдете до смещения FF54. Это место находится в районе страницы A-80 (точное положение зависит от даты издания и его полиграфического оформления). В приложении A "Технического руководства по IBM PC/XT" (PC/XT Tecnical Reference Manual) отдельно приводится листинг той части системы BIOS, которая относится к адаптеру жесткого диска. Это программа содержится в ПЗУ; ее начальный адрес имеет номер блока C800 (см. рис. 6.1).

В табл. 6.1 прерывания системы BIOS делятся на пять групп:

1. Векторы прерываний микропроцессора 8088 (типы 0 – 7H).
2. Векторы прерываний микроконтроллера 8259 (типы 8H – 0FH).

Таблица 6.1. **Векторы прерывания системы BIOS**

Тип прерывания	Назначение	Начальное присваивание
	Векторы прерываний микропроцессора 8088	
0	Деление на нуль	Зависит от версии DOS
1	Пошаговый режим исполнения	Команда IRET
2	Немаскируемое прерывание	NMI_INT (F000:E2C3 для PC, F000:F85F для XT)
3	Точка приостанова	Команда IRET
4	Переполнение	Команда IRET
5	Печать содержимого экрана	PRINT_SCREEN (F000:FF54)
6	Зарезервирован	–
7	Зарезервирован	–
	Векторы прерываний микроконтроллера 8259	
8	Системный таймер 8253	TIMER_INT (F000:FEA5)
9	Клавиатура	KB_INT (F000:E987)
A	Зарезервирован	–
B	Зарезервирован	–
C	Зарезервирован	–
D	В PC не используется, жесткий диск в XT	HD_INT (C800:0760)
E	Гибкий диск	DISK_INT (F000:EF57)
F	Зарезервирован	–
	Входные точки системы BIOS	
10	Обмен данными с дисплеем	VIDEO_IO (F000:F065)
11	Чтение конфигурации системы	EQUIPMENT (F000:F84D)
12	Возвращение объема памяти	MEMORY_SIZE_DET (F000:F841)
13	Обмен данными с диском	DISKETTE_IO (F000:EC59) для PC DISK_IO (C800:0256) для XT
Примечание. См. также прерывание типа 40.		
14	Обмен данными через последовательный порт	RS232_IO (F000:E739)
15	Обмен данными с кассетным магнитофоном	CASSETTE_IO (F000:F859)
16	Обмен данными с клавиатурой	KEYBOARD_IO (F000:E82E)
17	Обмен данными с принтером	PRINTER_IO (F000:EFD2)
18	Кассетный Бейсик	(F600:0000)
19	Сброс в начальное состояние	BOOT_STRAP (F000:E6F2 для PC, C800:0186 для XT)
1A	Время дня	TIME_OF_DAY (F000:FE6E)
40	Обмен данными с гибким диском для IBM XT	DISKETTE_IO (F000:EC59)

Тип прерывания	Назначение	Начальное присваивание
	Вызовы процедур пользователя	
18	Клавиша прерывания	Команда IRET
1C	Отсчет таймера	Команда IRET
	Указатель системных таблиц	
1D	Параметры изображения	VIDEO_PARMS (F000:F0A4)
1E	Параметры гибкого диска	DISK_BASE (F000:EFC7)
1F	Дополнительные графические символы	0:0
41	Параметры жесткого диска для IBM XT.	FD_TBL (C800:03E7)

3. Входные точки процедур системы BIOS (типы 10H – 1AH и 40H).
4. Вызовы процедур пользователя (типы 1BH и 1CH).
5. Указатели системных таблиц (типы 1DH, 1EH, 1FH и 41H).

В этом разделе мы опишем каждую группу прерываний и акцентируем внимание на тех прерываниях, которые могут Вам понадобиться в программах. Если не оговорено противное, то все утверждения справедливы как для IBM PC, так и для IBM XT.

ВЕКТОРЫ ПРЕРЫВАНИЙ МИКРОПРОЦЕССОРА 8088

Фирма Intel Corporation, производящая микропроцессоры 8088, требует, чтобы первые пять типов прерываний (0 – 4) имели одинаковые функции во всех системах, выполненных на базе микропроцессоров 8088 или 8086. И лишь одно прерывание этой группы (типа 5) может быть переприсвоено. Оно инициирует печать содержимого экрана на принтере.

ТИП 0 (ДЕЛЕНИЕ НА НУЛЬ)

Это прерывание инициируется, если при исполнении команды деления (DIV или IDIV) получается частное, не помещающееся в регистре, предназначенном для хранения результата. В этом случае работа программы прерывается и на экран выдается сообщение Divide overflow (переполнение при деление). После этого управление возвращается операционной системе DOS.

ТИП 1 (ПОШАГОВЫЙ РЕЖИМ ИСПОЛНЕНИЯ)

Это прерывание обеспечивает возможность пошагового исполнения программы в целях отладки. Операционная система DOS присваивает этому вектору адрес команды IRET (Interrupt return – возвратиться после обработки преры-

вания). Таким образом, команда INT 1 заставляет микропроцессор 8088 перейти к команде IRET, которая возвращает управление команде, следующей за командой INT 1. Фирма IBM не пользуется этим прерыванием, поскольку возможность пошагового исполнения программы обеспечивается командой T (Trace – трассировка) отладчика DEBUG.

ТИП 2 (НЕМАСКИРУЕМОЕ ПРЕРЫВАНИЕ)

С помощью команды CLI Вы можете обнулить флаг прерывания IF и тем самым заставить микропроцессор игнорировать все прерывания, кроме одного – прерывания типа 2; оно не может быть подавлено.

Средства расширения памяти IBM PC используют немаскируемое прерывание – NMI (Non-Maskable Interrupt) – для выдачи сообщений об ошибке в ячейке памяти. Программа обработки этого прерывания NMI_INT изображает на экране сообщение Parity Check 1 (если ошибка возникла на системной плате) или Parity Check 2 (если ошибка возникла на дополнительной плате). После этого она подавляет прерывания командой CLI и останавливает микропроцессор командой HLT.

Если в Вашей ЭВМ установлен математический сопроцессор 8087, то он использует прерывание NMI при выдаче сообщений об ошибках. Конечно, в этом случае Вам необходимо иметь дополнительное программное обеспечение, позволяющее различить, возникло прерывание из-за ошибки в ячейке памяти или из-за ошибки, обнаруженной сопроцессором 8087.

ТИП 3 (ТОЧКА ПРИОСТАНОВА)

Это прерывание позволяет продолжать исполнение программы до тех пор, пока микропроцессор 8088 не обнаружит указанный адрес остановки, или *точку приостанова*. Операционная система DOS делает вектор прерывания типа 3 указателем на команду IRET, но отладчик DEBUG временно изменяет его, если Вы указали параметр приостанова в команде G (Go – идти дальше).

ТИП 4 (ПЕРЕПОЛНЕНИЕ)

Это прерывание инициируется при исполнении микропроцессором 8088 команды прерывания при переполнении INTO. Операционная система DOS делает вектор прерывания типа 4 указателем на команду IRET, поскольку фирма IBM не может предугадать Ваши намерения в случае возникновения переполнения и предоставляет выбор Вам.

ТИП 5 (ПЕЧАТЬ СОДЕРЖИМОГО ЭКРАНА)

Последнее прерывание в этой группе (тип 5) обеспечивает вызов программы, входящей в состав BIOS. Программа обработки этого прерывания PRINT_SCREEN сохраняет текущее положение курсора, передает изображенную на экране информацию на принтер, а затем возвращает курсор на прежнее место. Следовательно, прерывание типа 5 вызывает тот же эффект, что и нажатие клавиши PrtSc на верхнем регистре, но дает возможность инициировать печать содержимого экрана из программы, а не с клавиатуры.

Ячейка памяти с адресом 50:5 содержит байт состояния принтера, который имеет следующие значения:

Во время печати ячейка 50:5 содержит 1;
если выдача на принтер завершилась успешно, то ячейка 50:5 содержит 0;
если во время печати возникла ошибка, то ячейка 50:5 содержит 0FFH.

ВЕКТОРЫ ПРЕРЫВАНИЯ МИКРОКОНТРОЛЛЕРА 8259

Микросхема 8259 (контроллер прерываний) располагается на системной плате и воспринимает сигналы запроса на прерывание (т. е. маскируемые прерывания) от восьми различных устройств системы. Действуя как электронный регулировщик движения, она передает микропроцессору 8088 сигнал запроса и код, идентифицирующий устройство. В персональной ЭВМ IBM PC к микроконтроллеру 8259 подсоединены только системный таймер 8253, клавиатура и контроллер гибкого диска. В персональной ЭВМ IBM XT к нему подключен также контроллер жесткого диска.

ТИП 8 (СИСТЕМНЫЙ ТАЙМЕР 8253)

Системный таймер 8253 представляет собой микросхему, установленную на системной плате и отсчитывающую время. Ее можно использовать для определения промежутка времени между двумя событиями и для генерации пауз. Таймер 8253 вызывает прерывание типа 8 каждые 0,0549254 с, т.е. он прерывает микропроцессор 8088 примерно 18,2 раза в секунду.

Программа обработки прерывания типа 8 TIMER_INT регистрирует прерывания от таймера 8253, пока прерывания разрешены (т. е. пока флаг прерываний IF равен 1), поэтому ее счетчиком можно пользоваться для регистрации времени дня. Счетчик времени представляет собой 32-битовое значение, расположенное в двух 16-битовых ячейках TIMER_LOW (0040:006C) и TIMER_HIGH (0040:006E). Программа TIMER_INT, кроме того, вызывает прерывание типа 1C при каждом отсчете таймера.

Система BIOS делает вектор прерывания типа 1C указателем на команду IRET, поэтому Вам придется изменить этот вектор, если Вы хотите, чтобы прерывание вызывало какую-либо полезную деятельность. Мы обсудим некоторые варианты использования прерывания типа 1C в подразделе "Вызовы процедур пользователя" и покажем, как устанавливать и считывать показания счетчика времени при обсуждении прерывания типа 1A (время дня) в подразделе "Входные точки системы BIOS".

ТИП 9 (КЛАВИАТУРА)

Это прерывание инициируется, если Вы нажимаете или отпускаете клавишу. Во всех практических ситуациях прерывание типа 9 можно рассматривать как системное. Мы обсудим более полезное прерывание от клавиатуры (типа 16) в подразделе "Входные точки системы BIOS".

ПРЕРЫВАНИЯ ТИПОВ D И E (ЖЕСТКИЙ ДИСК И ГИБКИЙ ДИСК)

Эти прерывания система BIOS получает в тех случаях, когда дисководу требуется выполнить обмен данными с памятью. Как и в случае прерывания типа 9, рассматривайте прерывания типов D и E как системные. Мы обсудим более полезное прерывание от дисковода типа 13 в следующем подразделе.

Большинство описываемых ниже прерываний выполняют функции ввода и вывода данных; они позволяют инициировать передачу данных периферийному устройству системы и обратно. Другие прерывания этой группы позволяют получать сведения о конфигурации системы и об объеме установленной в ней памяти, а также устанавливать и считывать время дня.

ТИП 10 (ОБМЕН ДАННЫМИ С ДИСПЛЕЕМ)

Это прерывание обеспечивает выполнение любой из 16 различных операций с дисплеем ЭВМ. Операция выбирается в зависимости от значения регистра AH. Программа обработки прерывания типа 10 VIDEO_IO начинает работу с загрузки начального адреса *буфера дисплея* (блока памяти, содержащего коды изображаемых на экране символов) в регистр дополнительного сегмента ES. Если в Вашей IBM PC установлена плата адаптера цветного графического монитора, то буфер дисплея имеет длину 16 Кбайт и начинается с ячейки B8000. Если в ней установлена плата адаптера монохроматического дисплея и принтера, то буфер имеет длину 4 Кбайт и начинается с ячейки B0000.

Загрузив нужное значение в регистр ES, программа VIDEO_IO выполняет требуемую операцию ввода-вывода. В табл. 6.2 перечислены эти операции и указаны используемые ими регистры. Все операции можно разделить на пять групп:

Процедуры интерфейса дисплея устанавливают режим изображения, ограничивают диапазон строк, по которым может передвигаться курсор, устанавлива-

Таблица 6.2. **Видеооперации ввода-вывода, инициируемые прерыванием типа 10**

Регистр AH	Операция	Дополнительные входные регистры		Выходные[1] регистры
Процедура интерфейса дисплея				
0	Задание режима изображения	(AL) = 0	40 × 25 ч/б[2], текстовый режим (по умолчанию)	Не используется
		(AL) = 1	40 × 25 цветной, текстовый режим	
		(AL) = 2	80 × 25 ч/б, текстовый режим	
		(AL) = 3	80 × 25 цветной, текстовый режим	
		(AL) = 4	320 × 200 цветной, графический режим	
		(AL) = 5	320 × 200 ч/б, графический режим	
		(AL) = 6	640 × 200 ч/б, графический режим	

[1] Наряду с возвращением значений в перечисленных здесь регистрах эти процедуры сохраняют значение регистров CS, SS, DS, ES, BX, CX и DX. Значения всех остальных регистров следует считать уничтоженными.
[2] Сокращение ч/б означает черно-белый. — *Прим. перев.*

Регистр AH	Операция	Дополнительные входные регистры	Выходные регистры
1	Задание горизонтального окна	CH, биты 0 – 4 = начальная строка окна CH, биты 5 – 7 = 0 CL, биты 0 – 4 = последняя строка окна CL, биты 5 – 7 = 0	Не используются
2	Перемещение в заданную позицию	(DH, DL) = (строка, столбец) (0,0) = левый верхний угол экрана (BH) = номер страницы (0 для графического режима)	Не используются
3	Чтение положения курсора	(BH) = номер страницы (0 для графического режима)	(DH, DL) = строка, столбец курсора (CH, CL) = текущий режим курсора
4	Чтение положения светового пера	Отсутствуют	(AH) = 0 переключатель светового пера не установлен в правильное положение или не нажата соответствующая кнопка (AH) = 1 в регистрах правильные значения координат светового пера (DH, DL) = строка, столбец (CH) = строка растра (0 – 199) (BX) = номер вертикальной линии (0 – 319) или (0 – 639)
5	Задание новой активной страницы дисплея (текстовый режим)	(AL) = новый номер страницы (0 – 7 для режимов 0 и 1, 0 – 3 для режимов 2 и 3)	Не используются
6	Прокрутка активной страницы вверх	(AL) = число строк; строки, появляющиеся внизу окна, заполняются пробелами (AL) = 0 заполняет пробелами все окно	Не используются

Регистр AH	Операция	Дополнительные входные регистры	Выходные регистры
		(CH, CL) = строка, столбец верхнего левого угла прокручиваемого окна (DH, DL) = строка, столбец нижнего правого угла прокручиваемого окна (BH) = атрибут, используемый при изображении пробельной строки	
7	Прокрутка активной страницы вниз	(AL) = число строк; строки, появляющиеся вверху окна, заполняются пробелами (AL) = 0 заполняет пробелами все окно (CH, CL) = строка, столбец верхнего левого угла прокручиваемого окна (DH, DL) = строка, столбец нижнего правого угла прокручиваемого окна (BH) = атрибут, используемый при изображении пробельной строки	Не используются
Процедуры обработки символов			
8	Чтение символа, находящегося в текущей позиции курсора, и его атрибута	(BH) = страница дисплея (текстовые режимы)	(AL) = считанный символ, (AH) = атрибут этого символа (текстовые режимы)
9	Запись символа и нового атрибута в текущую позицию курсора	(BH) = страница дисплея (текстовые режимы) (BL) = атрибут символа (текстовый режим) или цвет символа (графический режим) (CX) = счетчик записываемых символов (AL) = записываемый символ	Не используются

Регистр AH	Операция	Дополнительные входные регистры	Выходные регистры
10	Запись символа без изменения атрибута в текущую позицию курсора	(BH) = страница дисплея (текстовые режимы) (CX) = счетчик записываемых символов (AL) = записываемый символ	Не используются
Графический интерфейс			
11	Задание палитры цветов (разрешение 320 × 200)	(BH) = номер цвета на палитре (0 – 127) (BL) = значение цвета, присваиваемое этому номеру	Не используются
12	Изображение точки	(DX) = номер строки растра (CX) = номер вертикальной линии (AL) = значение цвета; если бит 7 регистра AL равен 1, то к этому значению и текущему значению цвета точки применяется операция OR	Не используются
13	Чтение точки	(DX) = номер строки растра (CX) = номер вертикальной линии	(AL) = считанная точка
Процедура ASCII-телетайпа для вывода			
14	Вывод символа на экран и перемещение курсора в следующую позицию	(AL) = выводимый символ (BL) = основной цвет (графический режим) (BH) = номер страницы (текстовый режим)	Не используются
Чтение видеостатуса			
15	Чтение текущего видеостатуса	Не используются	(AL) = текущий режим – см. (AH) = 0 для разъяснения (AH) = число столбцов на экране (BH) = текущая активная страница дисплея

ют и считывают положение курсора, считывают положение светового пера и управляют активной страницей дисплея.

Процедуры обработки символов передают символы на экран и обратно.

Процедуры графического интерфейса изображают и считывают точки графического изображения и изменяют цвета.

Процедура ASCII-телетайп выводит символ на экран и затем продвигает курсор.

Процедура чтения видеостатуса возвращает текущий режим, ширину экрана и страницу изображения.

У Вас могут возникнуть вопросы по поводу некоторых терминов, употребляемых в табл. 6.2 (например, "атрибут" и "значение цвета"), и по поводу программ обработки прерываний. Пожалуйста, потерпите, мы постараемся дать на них ответ при обсуждении графического режима в гл. 7.

ТИП 11 (ЧТЕНИЕ КОНФИГУРАЦИИ СИСТЕМЫ)

Прерывание типа 11 обеспечивает получение информации о том, какое оборудование входит в состав ЭВМ (на основании положении микропереключателей).

Эта информация возвращается в регистре AX (рис. 6.2). Например, в IBM PC/XT с монохроматическим дисплеем и принтером, не имеющей модемного или игрового устройства, по прерыванию типа 11 в регистре AX будет возвращено значение 4223H.

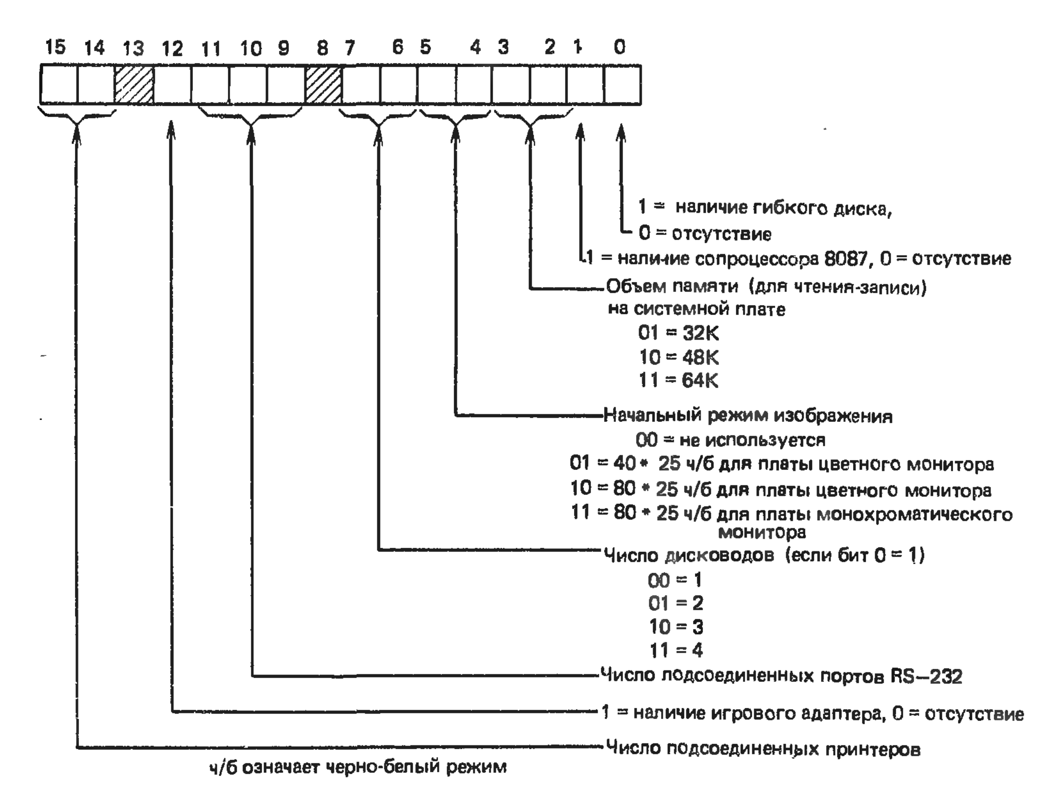


Рис. 6.2. Информация о составе оборудования ЭВМ, возвращаемая в регистре AX при прерывании 11

Конечно, эта информация бесполезна, если Вы пишете программы для собственной ЭВМ (Вы и так знаете, какое оборудование входит в ее состав). Однако она существенна при промышленной разработке программного обеспечения. Используя содержимое регистра AX, Вы можете настраивать программное обеспечение так, чтобы оно могло исполняться на различных системах. Это можно сделать за счет вызова тех или иных системно-зависимых процедур в зависимости от содержимого регистра AX.

Обратите особое внимание на то, что по биту 1 регистра AX можно определить, установлен ли в Вашей ЭВМ математический сопроцессор 8087. Если Вы написали программу, которая пользуется командами сопроцессора 8087, то используйте прерывание типа 11 для того, чтобы сообщить ей, исполнять эти команды или обходить их.

ТИП 12 (ОБЪЕМ ПАМЯТИ)

Прерывание типа 12 обеспечивает получение информации о том, сколько блоков в 1 Кбайт насчитывает ЗУПВ Вашей ЭВМ (на основании положений микропереключателей). Число блоков возвращается в регистре AX.

ТИП 13 (ОБМЕН ДАННЫМИ С ДИСКОМ)

Команды операционной системы DOS позволяют Вам выполнять манипуляции с *файлами*. Прерывание типа 13 позволяет получить более широкие возможности доступа к дисковой информации за счет выполнения операций с отдельными *дорожками* или *секторами*. Система BIOS IBM PC делает вектор прерывания типа 13 указателем на программу обслуживания гибкого диска DISKETTE_IO; система BIOS IBM XT делает его указателем на программу обслуживания жесткого диска DISK_IO. (В систему BIOS IBM XT входит и программа DISKETTE_IO, но для доступа к ней надо пользоваться прерыванием типа 40, а не 13.)

Прерывание типа 13 обеспечивает выполнение многих функций, полезных при разработке дисковых утилит или схемы защиты от копирования. Предупреждаем, однако, что все эти операции предназначены для людей с закаленными нервами. Прежде чем пустить их в ход, не мешает серьезно взвесить возможные последствия. За деталями обратитесь к техническому руководству фирмы IBM.

ТИП 14 (ОБМЕН ДАННЫМИ ЧЕРЕЗ ПОСЛЕДОВАТЕЛЬНЫЙ ПОРТ)

Это прерывание позволяет Вам передавать и получать информацию через последовательный порт ввода-вывода IBM PC. Детальные сведения можно найти в описании последовательного асинхронного адаптера в техническом руководстве фирмы IBM.

ТИП 15 (ОБМЕН ДАННЫМИ С КАССЕТНЫМ МАГНИТОФОНОМ)

Это прерывание предусмотрено для работы с кассетным магнитофоном. Так как на самом деле ни у одной IBM PC нет кассетного магнитофона, то нам его обсуждать незачем.

ТИП 16 (ОБМЕН ДАННЫМИ С КЛАВИАТУРОЙ)

Это основное прерывание системы BIOS, используемое для обмена данными с клавиатурой. Однако из-за большого числа выполняемых им функций

Таблица 6.3. **Операции по обмену данными с принтером, инициируемые прерыванием типа 17**

Регистр АН	Операция	Дополнительные входные регистры	Выходные регистры
0	Печать символа	(AL) = символ (DX) = номер принтера (0 – 2)	(AH) = состояние операции (см. рис. 6.3)
1	Сброс принтера в начальное состояние	(DX) = номер принтера (0 – 2)	Тот же, что и в процедуре печати
2	Чтение состояния принтера	(DX) = номер принтера (0 – 2)	Тот же, что и в процедуре печати

и значительной гибкости клавиатурной системы ввода-вывода IBM PC мы отложим обсуждение прерывания типа 16 до разд. 6.4, в котором дадим ему настолько детальное описание, насколько оно его заслуживает.

ТИП 17 (ОБМЕН ДАННЫМИ С ПРИНТЕРОМ)

Это прерывание позволяет Вам работать с тремя принтерами (если Вы настолько богаты). Программа обработки прерывания типа 17 (PRINTER_IO) позволяет напечатать символ, инициировать порт, к которому подсоединен принтер, или прочитать его состояние (табл. 6.3). Эти операции возвращают байт состояния в регистре АН (рис. 6.3), но не воздействуют ни на какие другие регистры.

ТИП 18 (КАССЕТНЫЙ БЕЙСИК)

Это прерывание вызывает Кассетный Бейсик, содержащийся в ПЗУ.

ТИП 19 (СБРОС В НАЧАЛЬНОЕ СОСТОЯНИЕ)

Это прерывание заставляет ЭВМ заново загрузить систему с диска. Если в дисководе А IBM PC не оказалось диска или обнаружилась ошибка в работе дисковода либо дискового контроллера, то ЭВМ загружает Кассетный Бейсик с помощью прерывания типа 18.

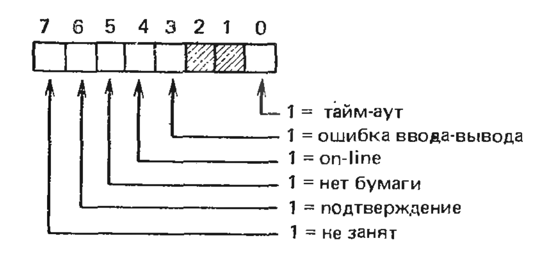


Рис. 6.3. Байт состояния принтера

Напомним, что прерывания типа 8 управляют счетчиком системных часов и инициируются встроенным системным таймером 8253. Эти прерывания происходят примерно 18,2 раза в секунду.

Прерывание типа 1А позволяет установить новое значение этого счетчика "времени дня" (32-битового числа без знака) или прочитать его текущее значение. С его помощью можно определить длительность событий при исполнении Вашей программы. Например, если Вы обнулите счетчик, исполните программу, а затем прочитаете значение счетчика, то результат покажет, сколько времени (в отсчетах таймера) заняло исполнение программы у микропроцессора 8088.

Как обычно, значение регистра АН определяет функцию, выполняемую прерыванием. Для *установки значения счетчика времени* загрузите 1 в регистр АН, а старшие и младшие 16 битов нового значения счетчика – в регистры СХ и DX соответственно. Для *считывания значения счетчика времени* загрузите 0 в регистр АН, IBM PC возвратит старшие и младшие 16 битов счетчика в регистр СХ и DX соответственно. Далее содержимое регистра AL укажет Вам, прошло ли 24 часа с момента последнего считывания счетчика. Если 24 часа еще не прошло, то содержимое регистра AL равно нулю: в противном случае оно отлично от нуля.

Так как прерывание типа 1А возвращает значение счетчика в отсчетах таймера, то оно полезно только для выполнения очень точных измерений интервалов времени. В разд. 6.3 мы обсудим прерывания операционной системы DOS, которые позволят Вам устанавливать и считывать время в часах, минутах, секундах и сотых долях секунды. Тем не менее прерывание 1А может быть полезно для такого приложения, как генерация случайных чисел.

ГЕНЕРАЦИЯ СЛУЧАЙНЫХ ЧИСЕЛ

Так как значения ячеек счетчика времени TIMER_LOW (0040:006C) и TIMER_HIGH (0040:006E) постоянно изменяются, то они (особенно содержимое ячейки TIMER_LOW) могут интерпретироваться как *псевдослучайные* числа! Так как значения ячеек счетчика меняются не произвольно, а постоянно возрастают, то они не порождают настоящих случайных чисел. Но поскольку значение времени увеличивается 18,2 раза в секунду, оно достаточно *случайно* для большинства приложений.

Будучи 16-битовой, ячейка TIMER_LOW может содержать значения от 0 до 65 535. Для генерации случайного числа в интервале от 0 до 51 (что может потребоваться при разработке программы игры в карты) надо прочитать текущее значение ячейки TIMER_LOW и разделить его на 52. Тем самым получится *остаток* в интервале от 0 до 51.

В примере 6.1 показана процедура RAND_51, которая возвращает в регистре АН значение от 0 до 51. Обратите внимание, что во избежание переполнения при делении мы обнуляем три старших бита ячейки TIMER_LOW. Если бы мы этого не сделали, то деление вызывало бы переполнение всякий раз, когда содержимое ячейки TIMER_LOW превышало бы десятичное число 13260 (т. е. 52 × 255).

ПРИМЕР 6.1. ГЕНЕРАЦИЯ СЛУЧАЙНОГО ЧИСЛА В ИНТЕРВАЛЕ ОТ 0 ДО 51

```
; Эта процедура использует ячейку TIMER_LOW, значение которой
; возвращается по прерыванию 1А, для генерации псевдослучайного
; числа в диапазоне от 0 до 51. Это число возвращается в регистре
```

```
; AH. Значения других регистров не изменяются .
;
RAND_51  PROC   ,
         PUSH  CX          ;Сохранить регистры, изменяемые прерыванием 1A
         PUSH  DX
         PUSH  AX
         MOV   AH,0        ;Считать показания таймера
         INT   1AH
         MOV   AX,DX       ;Переместить младшую часть счетчика в AX
         AND   AX,1FFFH    ; и удалить из него старшие 3 бита
         MOV   CL,52       ;Разделить младшую часть счетчика на 52
         DIV   DL
         POP   DX          ;Восстановить AL
         MOV   AL,DL
         POP   DX          ;Восстановить DX и CX,
         POP   CX
         RET               ; и вый-· из процедуры
RAND_51  ENDP
```

ВЫЗОВЫ ПРОЦЕДУР ПОЛЬЗОВАТЕЛЯ

Эти два прерывания позволяют Вам расширить возможности системы:

ТИП 1B (КЛАВИША ПРЕРЫВАНИЯ)

Это прерывание инициируется при нажатии клавиш Ctrl-Break. Оно вызывается во время обработки микропроцессором 8088 прерывания типа 9, инициируемого клавиатурой ЭВМ. Система BIOS делает вектор прерывания типа 1B указателем на команду IRET, но операционная система DOS заменяет его на адрес программы обработки прерывания типа 23 (адрес выхода по Ctrl-Break). Прерывание типа 23, относящееся к группе прерываний операционной системы DOS, описано в разд. 6.3.

ТИП 1C (ОТСЧЕТ ТАЙМЕРА)

Это прерывание вызывается программой TIMER_INT, обрабатывающей прерывание типа 8 (инициируемое системным таймером 8253). Поэтому прерывание типа 1C происходит примерно 18,2 раза в секунду, как и прерывание типа 8 (при условии, что прерывания не подавлены).

Прерывания типа 1C позволяют Вам дать микропроцессору 8088 определенную дополнительную работу, которую он будет выполнять при каждом отсчете таймера 8253. Система BIOS делает вектор прерывания типа 1C указателем на команду IRET, так что прерывание типа 1C не инициирует никакой полезной работы (если только Вы не измените этот указатель).

Прерыванием типа 1C можно пользоваться для того, чтобы периодически изображать десятичное значение счетчика времени и получать электронные часы. А можно написать такую программу обработки прерывания типа 1C, чтобы она проверяла счетчик времени и инициировала звуковой сигнал, как только счетчик достигнет определенного значения; тем самым Вы получите компьютеризованный будильник. Наверное, Вы сможете найти и другие применения для этого полезного прерывания.

Векторы прерываний типов 1D – 1F, а также 41 не связаны с программами обработки прерываний; они являются адресами системных таблиц. Адрес, соответствующий типу 1D, указывает на таблицу, используемую прерыванием типа 10. Адрес, соответствующий типу 1E, указывает на таблицу гибкого диска, используемую прерыванием типа 13 в IBM PC или прерыванием типа 40 в IBM XT. Адрес, соответствующий типу 41, указывает на таблицу жесткого диска, используемую прерыванием типа 13 в IBM XT.

Ячейка, соответствующая типу 1F, зарезервирована в качестве указателя на таблицу псевдографики, состоящую из специальных символов. Но так как этой таблицы в памяти нет, то система BIOS дает этому вектору начальное значение 0:0. Программа GRAFTABL.COM, работающая под управлением операционной системы DOS версии 3.0, обеспечивает появление дополнительного набора символов. Будучи загружена в систему, она делает вектор прерывания типа 1F указателем на таблицу размером в 1Кбайт, обеспечивающую 128 дополнительных символов для псевдографического режима. Каждый символ задан массивом, представляющим собой матрицу размером 8*8 битов (размещенную в 8 байтах), первый байт которой определяет верхнюю строку растра символа.

6.3. ПРЕРЫВАНИЯ ОПЕРАЦИОННОЙ СИСТЕМЫ DOS

Фирма IBM резервирует прерывания типов 20 – 3F для использования операционной системой DOS. Из них в настоящее время задействованы типы 20 – 27. Они описаны в табл. 6.4.

Большинство из этих прерываний полезны только для DOS, и мы не будем здесь на них останавливаться. (Если Ваша ЭВМ работает с операционной системой DOS версии 1.1 или 2.0, то детальное описание этих прерываний можно найти в приложении D руководства по дисковой операционной системе фирмы IBM. Если у Вас DOS версии 2.1 или позже, то см. техническое руководство по DOS.) Однако прерывание типа 21 (вызов функций) предоставляет Вам множество удобных возможностей взаимодействия с клавиатурой, дисплеем, принтером, диском и асинхронным последовательным устройством. Наиболее полезные функции, вызываемые с помощью прерывания типа 21, перечислены в табл. 6.5.

Таблица 6.4. **Прерывания операционной системы DOS**

Тип прерывания	Назначение
20	Завершение программы
21	Вызовы функций
22	Адрес завершения
23	Адрес выхода при обработке комбинации клавиш Ctrl-Break
24	Обработка критичных ошибок
25	Абсолютное чтение с диска
26	Абсолютная запись на диск
27	Завершение программы с сохранением ее в памяти
28-3F	Зарезервированы для DOS

Таблица 6.5. **Вызовы функций, инициируемые прерыванием типа 21**

Регистр AH	Операция	Дополнительные входные регистры	Выходные регистры
Функции для работы с клавиатурой			
1	Ожидание набора символа на клавиатуре и последующее изображение его на экране (с проверкой на Ctrl-Break[1])	Не используются	(AL) = символ
6	Чтение символа с клавиатуры (без проверки на Ctrl-Break [1])	(DL) = 0FFH	(AL) = очередной символ, если буфер клавиатуры не пуст; (AL) = 0, если буфер клавиатуры пуст
7	Ожидание набора символа на клавиатуре без последующего его изображения (без проверки на Ctrl-Break [1])	Не используются	(AL) = символ
8	То же, что функция 7, но с проверкой на Ctrl-Break	Не используются	(AL) = символ
A	Чтение клавиатурной строки в буфер	(DS:DX) = адрес буфера Первый байт буфера = размер буфера	Второй байт буфера = число фактически прочитанных символов
B	Чтение состояния клавиатуры	Не используются	AL = 0FFH, если клавиатурная строка пуста, AL = 0, если она содержит хотя бы один символ
C	Опустошение буфера клавиатуры и вызов функции для работы с клавиатурой	(AL) = номер функции для работы с клавиатурой	В соответствии с вызываемой функцией
Функции для работы с дисплеем			
2	Изображение символа (с проверкой на Ctrl-Break)	(DL) = символ	Не используются

[1] Некоторые комбинации клавиш генерируют "расширенные коды", и для их чтения может потребоваться два вызова функций. Детали см. в разд. 6.4.

Регистр АН	Операция	Дополнительные входные регистры	Выходные регистры
5	Печать символа	(DL) = символ	Не используются
6	Изображение символа (без проверки на Ctrl-Break)	(DL) = символ	Не используются
9	Изображение строки	(DS:DX) = адрес строки, которая должна заканчиваться знаком $	Не используются
Функции для обмена данными с асинхронным последовательным устройством			
3	Ожидание ввода символа через асинхронное последовательное устройство ввода	Не используются	(AL) = символ
4	Вывод символа на асинхронное последовательное устройство	(DL) = символ	Не используются
Функции управления файлами			
D	Сброс текущего дисковода в начальное состояние	Не используются	Не используются
E	Задание нового текущего дисковода	(DL) = номер дисковода (0 = A, 1 = B, 2 = C)	(AL) = число дисководов (2 в случае одного дисковода)
2E	Задание режима проверки	(DL) = 0 (AL) = 0 для отключения проверки (AL) = 1 для включения проверки	Не используются

Примечание. Описание других функций с номером в диапазоне от F до 2F, предназначенных для работы с диском в операционной системе DOS версии 1.1, можно найти в "Техническом руководстве по операционной системе DOS". Пользователи операционной системы DOS версии 2 должны вместо них использовать функции расширенного управления файлами.

Регистр AH	Операция	Дополнительные входные регистры	Выходные регистры
Функции для работы с датами и временем			
2A	Чтение системной даты	Не используются	(CX) = год (1980 – 2099) (DH) = месяц (1 – 12) (DL) = деиь (1 – 31)
2B	Установка системной даты	(CX, DX) = дата в том же формате, что и для функции 2A	(AL) = 0, если дата правильная (AL) = FF, если дата ошибочная
2C	Чтение системного времеии	Не используются	(CH) = часы (0 – 23) (CL) = мииуты (0 – 59) (DH) = секунды (0 – 59) (DL) = сотые доли секуиды (0 – 99)
2D	Установка системного времени	(CX, DX) = время в том же формате, что и для функции 2C	(AL) = 0, если время правильное (AL) = FF, если время ошибочное
Функции для работы с векторами прерываиий			
25	Установка вектора прерывания	(DS:DX) = новое значение вектора	Не используются
35	Чтение вектора прерывания	(AL) = номер (тип) прерывания	(ES:BX) = значение вектора
Функции для работы со справочниками файлов (только в операционной системе DOS версии 2)			

Примечание. Значения возвращаемых кодов ошибки см. в разделе 6.6.

39	Создание справочника файлов MKDIR	(DS:DX) = адрес ASCIIZ-строки с именем справочника	Может возвращаться код ошибки 3 или 5

Регистр AH	Операция	Дополнительные входные регистры	Выходные регистры
3A	Удаление справочника файлов RMDIR	(DS:DX) = адрес ASCIIZ-строки с именем справочника	Может возвращаться код ошибки 3 или 5
3B	Смена текущего справочника файлов (CHDIR)	(DS:DX) = адрес ASCIIZ-строки с именем нового справочника файлов	Может возвращаться код ошибки 3
47	Чтение имени текущего справочника файлов	(DL) = номер дисковода (0 = текущий, 1 = A и т. д.) (DS:DI) = адрес буфера размером 64 байта	(DS:SI) = адрес ASCIIZ-строки Может возвращаться код ошибки 15

Функции расширенного управления файлами (только в операционной системе DOS версии 2)

Примечание. Значения возвращаемых кодов ошибки см. в табл. 6.6.

Регистр AH	Операция	Дополнительные входные регистры	Выходные регистры
36	Получение сведений о свободном месте на диске	(DL) = номер дисковода (0 = текущий, 1 = A и т. д.)	(AX) = 0FFFFH, если обнаружена ошибка (AX) = число секторов в кластере (BX) = число свободных кластеров (DX) = общее число кластеров (CX) = число байтов в секторе
3C	Создание файла	(DS:DX) = адрес ASCIIZ-строки (CX) = атрибут файла	(AX) — логический номер файла Могут возвращаться коды ошибок 3, 4 или 5
3D	Открытие файла	(DS:DX) = адрес ASCIIZ-строки (AL) = 0 при открытии для чтения (AL) = 1 при открытии для записи (AL) = 2 при открытии для чтения и записи	(AX) = логический номер файла Могут возвращаться коды ошибок 2, 4, 5 или 12

Регистр AH	Операция	Дополнительные входные регистры	Выходные регистры
3E	Закрытие логического номе́ра файла	(BX) = логический номер файла	Может возвращаться код ошибки 6
3F	Чтение файла или ввод данных с устройства	(BX) = логический номер файла (CX) = число считываемых байтов (DS:DX) = адрес буфера	(AX) = число фактически считанных байтов (AX) = 0 при попытке чтения за концом файла Могут возвращаться коды ошибок 5 или 6
40	Запись в файл или вывод данных на устройство	(BX) = логический номер файла (CX) = число записываемых байтов (DS:DX)= адрес буфера	(AX) = число фактически записанных байтов Могут возвращаться коды ошибок 5 или 6
41	Удаление файла	(DS:DX) = адрес ASCIIZ-строки	Могут возвращаться коды ошибок 2 или 5
43	Чтение атрибута файла	(AL) = 0 (DS:DX) = адрес ASCIIZ-строки с именем файла	(CX) = атрибут. Могут возвращаться коды ошибок 2 или 5
43	Установка нового атрибута файла	(AL) =1 (DS:DX) = адрес ASCIIZ-строки с именем файла (CX) = атрибут	Могут возвращаться коды ошибок 2 или 5
54	Чтение режима проверки	Не используются	(AL) = 0, если проверка выключена (AL) = 1, если проверка включена
56	Переименование файла	(DS:DX) = адрес ASCIIZ-строки со старым именем файла (ES:DI) = адрес ASCIIZ-строки с новым именем файла	Могут возвращаться коды ошибок 3, 5 или 17

Таблица 6.6. **Коды ошибок при вызовах функций операционной системы DOS версии 2**

Код	Значение
1	Ошибочный номер функции
2	Файл не найден
3	Путь к файлу не найден
4	Слишком много открытых файлов (не осталось свободных логических номеров)
5	Доступ не разрешен
6	Ошибочный логический номер файла
7	Управляющий блок памяти разрушен
8	Недостаточно памяти
9	Ошибочный адрес блока памяти
10	Ошибочная аппаратная среда
11	Ошибочный формат
12	Ошибочный код доступа
13	Ошибочные данные
14	Ошибочное имя дисковода
15	Попытка удалить текущий справочник файлов
16	Не то устройство
17	Превышен предел числа файлов

Функции с номерами 0 – 2E одинаковы для всех версий DOS после 1.0, а функции с номерами 2F и выше имеются в DOS версии 2.0 или более поздней. Многие из этих новых функций (см. подразделы "Функции для работы со справочниками файлов" и "Функции расширенного управления файлами") возвращают флаг CF равным 0, если операция выполнена успешно, и равным 1 в случае возникновения ошибки. При CF = 1 код ошибки возвращается ими в регистре AX. В табл. 6.6 описаны значения этих кодов.

ТИП 21 (ВЫЗОВЫ ФУНКЦИЙ)

ФУНКЦИИ ДЛЯ РАБОТЫ С КЛАВИАТУРОЙ

Эти функции достаточно просты; они считывают либо отдельные набираемые на клавиатуре символы в регистр AL, либо последовательность символов *(строку)* в память. Если только Вам не приходится иметь дело с некоторыми необычными комбинациями клавиш (подробнее об этом в разд. 6.4), то Вы найдете эти функции удобными для применения.

В диалоговых программах от пользователя нередко требуется дать ответ на приглашение к вводу или сделать выбор из меню вводом одной буквы или цифры. Пусть, например, Ваша программа изображает сообщение, в котором от пользователя требуется нажать клавишу либо с буквой Д, либо с буквой Н (для продолжения или прекращения работы). Ввод Д заставляет программу перейти к группе команд, помеченных меткой YES, а ввод Н – к команде с меткой NO. При нажатии

любой другой клавиши программа снова должна ожидать ввода либо буквы Д, либо буквы Н. Эту задачу выполняет следующий фрагмент:

```
GET_KEY:  MOV   AH,1        ;Считать символ
          INT   21H
          CMP   AL,'Д'      ;Он равен Д?
          JE    YES         ; Если да, то перейти к метке YES
          CMP   AL,'Н'      ;Он равен Н?
          JE    NO          ; Если да, то перейти к метке NO
          JNE   GET_KEY     ; В противном случае ждать ввода Д или Н
```

Данный фрагмент распознает только прописные буквы Д и Н, но пользователи склонны набирать символы, не нажимая клавишу верхнего регистра. Чтобы программа воспринимала ответы, набранные строчными буквами, добавьте дополнительные команды сравнения CMP AL, 'д' и CMP AL, 'н'.

Во многих приложениях требуется, чтобы пользователь ввел строку символов, например фамилию и адрес. Для этой цели служит функция А. Чтобы воспользоваться ею, Вы должны зарезервировать в сегменте данных место для строки. А именно требуется выделить блок байтов, длина которого на два байта превышает максимальное число символов в строке. Первый байт этого блока должен задавать максимальную длину строки. Например, чтобы зарезервировать место для строки из 50 символов, надо в сегменте данных указать оператор

```
USER_STRING  DB 50,51 DUP(?)
```

Чтение строки выполняется командами

```
LEA DX,USER_STRING      ;Сделать DX указателем буфера
MOV AH,0AH              ;Прочитать
INT 21H                 ;строку
```

Функция А помещает счетчик фактически введенных символов (исключая возврат каретки) во второй байт строки и оставляет пару регистров DS:DX указателем на байт, содержащий максимальную длину строки. Во многих случаях желательно, чтобы счетчик символов был помещен в регистр CX, а пара регистров DS:DX указывала на первый символ строки. Эту работу выполняет процедура READ_KEYS из примера 6.2.

ФУНКЦИИ ДЛЯ РАБОТЫ С ДИСПЛЕЕМ

Имеются также функции, изображающие или печатающие находящийся в регистре AL символ, и функция, изображающая строку. Для последней функции 9 требуется, чтобы в конце строки стоял знак доллара. Поэтому для изображения приглашения "Пожалуйста, введите Вашу фамилию:" надо в сегменте данных указать оператор

```
PROMPT  DB 'Пожалуйста, введите Вашу фамилию: $'
```

а в программе – команды

```
LEA  DX,PROMPT
MOV  AH,9
INT  21H
```

ПРИМЕР 6.2. ЧТЕНИЕ СТРОКИ С КЛАВИАТУРЫ

```
; Эта процедура считывает до 50 ударов по клавишам. Она возвращает
; адрес прочитанной строки в регистрах DS:DX и счетчик символов в
; регистре CX. Значения других регистров не изменяются
;
; Поместите следующий оператор в сегмент данных
;
USER_STRING   DB 50,51 DUP(?)
;
; Ниже приводится фактическая процедура
;
READ_KEYS  PROC
      PUSH  AX
      LEA   DX,USER_STRING      ;Считать строку
      MOV   AH,0AH
      INT   21H
      SUB   CH,CH               ;Поместить счетчик символов в CX
      MOV   CL,USER_STRING+1
      ADD   DX,2                ;Сделать DX указателем на строку
      POP   AX
      RET
READ_KEYS  ENDP
```

Эти команды оставляют курсор в том месте, где Вы хотите видеть фамилию пользователя при наборе ее на клавиатуре: через один пробел от двоеточия.

Чтобы после выдачи строки символов курсор пререшел на начало следующей строки экрана (в случае, если выдается сообщение, а не приглашение к вводу), надо перед знаком $ вставить символы возврата каретки и перехода на следующую строку, например:

```
MESSAGE  DB  Операция сортировки завершена. ,0DH,0AH,'$'
```

Эти и другие управляющие символы мы обсудим в разд. 6.4.

ФУНКЦИИ УПРАВЛЕНИЯ ФАЙЛАМИ

Перечисленные в таб. 6.5 *функции управления* файлами составляют малую часть общего числа функций, доступных в любой версии операционной системы DOS. Кроме указанных в этой таблице функций, есть и другие, которые открывают, закрывают, удаляют файлы, переименовывают их и т. д. Однако при этом требуется выполнять операции над блоками управления файлами. FCB (File control block), которые могут оказаться довольно сложными. С появлением версии DOS 2.0 фирма IBM предоставила возможность выполнять эти действия с помощью гораздо более простых функций – так называемых функций расширенного управления файлами. Мы коснемся их несколько позже.

ФУНКЦИИ ДЛЯ РАБОТЫ С ДАТАМИ И ВРЕМЕНЕМ

Вряд ли Вам захочется изменять текущую дату из программы, написанной на языке ассемблера, но доступ к счетчику времени может оказаться желательным. Например, чтобы определить время исполнения процедуры, можно обнулить счетчик времени, вызвать процедуру и после возврата из нее прочитать значение счетчика. В результате Вы получите затраченное время. Следующий фрагмент вычисляет время исполнения процедуры SORT:

```
        SUB     CX,CX       ;Установить нулевое время
        SUB     DX,DX
        MOV     AH,2DH
        INT     21H
        CALL    SORT        ;Выполнить процедуру
        MOV     AH,2CH      ;Считать время выполнения
        INT     21H
```

Чтобы выполнить ту же операцию без переустановки системного счетчика времени, надо вычесть из конечного времени начальное, используя рабочие ячейки в сегменте данных для сохранения начального времени:

```
          MOV     AH,2CH      ;Считать начальное время
          INT     21H
          MOV     HRS,CH      ; и сохранить его
          MOV     MINS,CL
          MOV     SECS,DH
          MOV     HSECS,DL
          CALL    SORT        ;Выполнить процедуру
          MOV     AH,2CH      ;Считать текущее время
          INT     21H
          SUB     DL,HSECS    ;Вычислить разность
          JNC     SUB_SECS
          ADD     DL,100
          DEC     DH
SUB_SECS: SUB     DH,SECS
          JNC     SUB_MINS
          ADD     DH,60
          DEC     CL
SUB_MINS: SUB     CL,MINS
          JNC     SUB_HRS
          ADD     CL,60
          DEC     CH
SUB_HRS:  SUB     CH,HRS
          JNC     DONE
          ADD     CH,24
DONE:     RET
```

После каждого вычитания с помощью дополнительных команд производится преобразование результата в положительное число.

Функции для работы со временем особенно полезны для генерации пауз. Паузы могут понадобиться, если мы хотим извлечь звуки из динамика ЭВМ или добиться движения графических образов на экране дисплея. При извлечении звуков пауза задает длительность ноты; при движении графических образов она задает либо время, в течение которого они видны на экране, либо время ожидания перед появлением следующего образа.

Программы генерации пауз обычно выполняют следующую последовательность действий:

1. Считывают текущее время.

2. Добавляют к текущему времени заданный интервал, чтобы получить время конца паузы.

3. Корректируют время конца паузы так, чтобы часы не превышали 24, минуты и секунды – 60, а сотые доли секунды – 100. Эти операции включают в себя добавление избытков к более крупным единицам времени.

4. Циклически считывают значения счетчика времени до тех пор, пока текущее время не совпадет со временем конца паузы или не превысит его.

На рис. 6.4 показана блок-схема процедуры генерации паузы, длительность которой задается в минутах, секундах и сотых долях секунды. (Мы предполагаем,

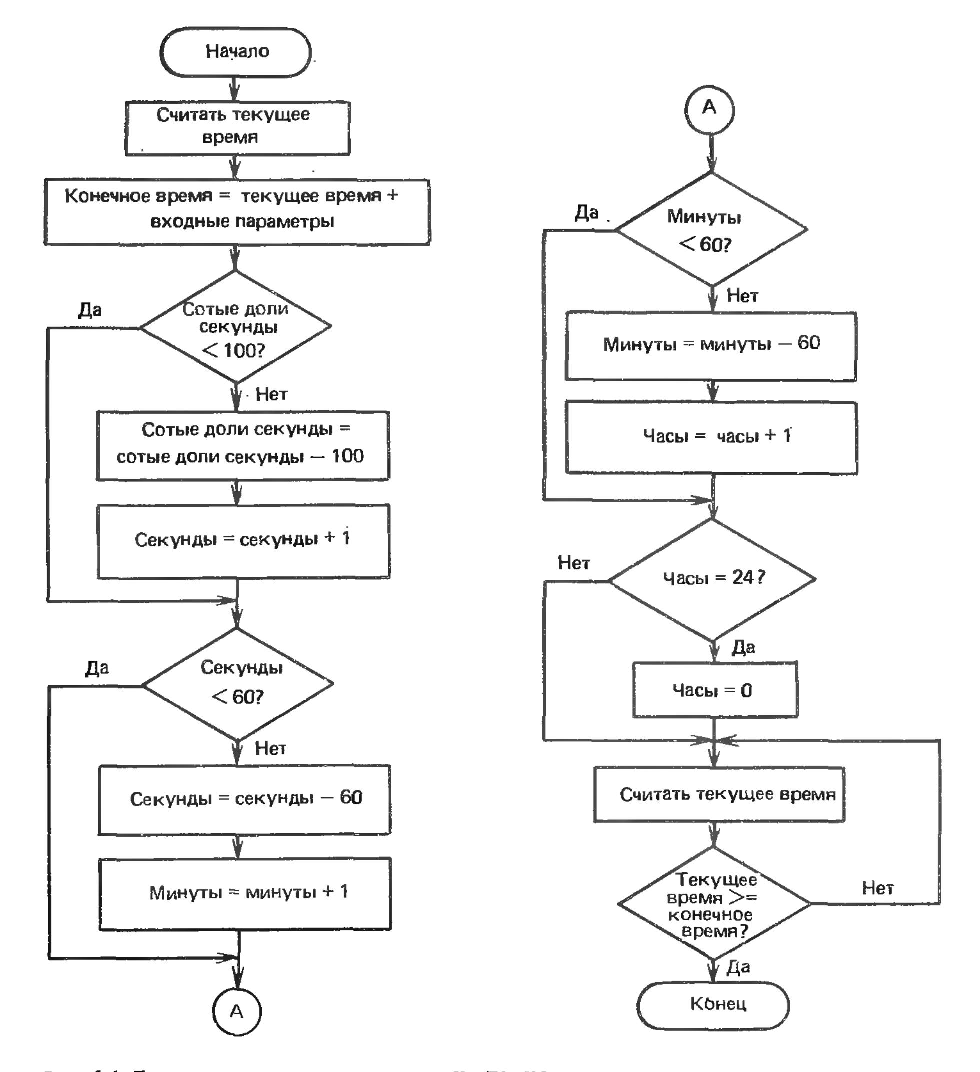


Рис. 6.4. Блок-схема процедуры генерации паузы

что Вам не захочется ждать более часа, хотя это и не возбраняется.) В примере 6.3 показана универсальная процедура генерации паузы DELAY, в которой использован указанный выше подход.

ПРИМЕР 6.3. ГЕНЕРАЦИЯ ЗАДАННОЙ ПАУЗЫ

```
; Эта процедура заставляет процессор ждать в течение заданного
; промежутка времени. Перед обращением к ней загрузите минуты в
; регистр AL, секунды - в регистр BH, а сотые доли секунды - в
; регистр BL
; Значения регистров не изменяются
;
DELAY   PROC
        PUSH    AX              ;Сохранить используемые регистры
        PUSH    BX
        PUSH    CX
```

```
        PUSH   DX
        MOV    AH,2CH          ;Считать текущее время
        INT    21H
;
; Сложить текущее время с входными значениями
;
        MOV    AH,CH           ;Часы
        ADD    AL,CL           ;Минуты
        ADD    BH,DH           ;Секунды
        ADD    BL,DL           ;Сотые доли секунды
;
; Скорректировать сотые доли, секунды, минуты и часы
;
        CMP    BL,100          ;Число сотых должно быть < 100
        JB     SECS
        SUB    BL,100
        INC    BH
SECS:   CMP    BH,60           ;Число секунд должно быть < 60
        JB     MINS
        SUB    BH,60
        INC    AL
MINS:   CMP    AL,60           ;Число минут должно быть < 60
        JB     HRS
        SUB    AL,60
        INC    AH
HRS:    CMP    AH,24           ;Число часов должно быть < 24
        JNE    CHECK
        SUB    AH,AH
;
; Ждать, пока не наступит вычисленное время
;
CHECK:  PUSH   AX              ;Считать время снова
        MOV    AH,2CH
        INT    21H
        POP    AX
        CMP    CX,AX           ;Сравнить часы и минуты
        JA     QUIT
        JB     CHECK
        CMP    DX,BX           ;Сравнить секунды и сотые доли секунды
        JB     CHECK
QUIT:   POP    DX              ;Восстановить значения регистров
        POP    CX
        POP    BX
        POP    AX
        RET
DELAY   ENDP
```

ФУНКЦИИ ДЛЯ РАБОТЫ С ВЕКТОРАМИ ПРЕРЫВАНИЙ

Функции 25 и 35 позволяют оперировать адресами, являющимися значениями векторов прерываний. Например, если требуется добавить к системе новую программу обработки прерывания NEW_INT и заставить ее реагировать на команду INT 60H, то надо воспользоваться командами

```
MOV   AL,60H               ;Поместить тип прерывания в AL
MOV   DX,SEG NEW_INT       ;Поместить номер блока в DS
MOV   DS,DX
MOV   DX,OFFSET NEW_INT    ; и смещение адреса в DX
MOV   AH,25H               ;Выбрать функцию 25
INT   21H                  ;Изменить вектор прерывания
```

Напомним, что программа обработки прерывания является просто процедурой, в конце которой указана команда IRET (вместо RET). Следовательно, программа NEW_INT имеет вид

```
NEW_INT  PROC FAR
    ...              (Команды программы обработки прерывания)
    ...
    IRET
NEW_INT  ENDP
```

ФУНКЦИИ ДЛЯ РАБОТЫ СО СПРАВОЧНИКАМИ ФАЙЛОВ

Эти функции позволяют прочитать имя текущего справочника файлов и вызвать из программы команды операционной системы DOS версии 2 MKDIR, RMDIR и CHDIR. По завершении каждая из них заносит во флаг CF либо 0 (успешная операция), либо 1 (ошибка). В последнем случае в регистре AX содержится код ошибки.

В каждом случае требуется задать строку символов ASCII, описывающую справочник файлов. Она может содержать имя дисковода (например, C:) и путь к файлу и должна заканчиваться байтом, имеющим значение 0 (а не код нуля в системе ASCII). Из-за наличия нулевого байта фирма IBM назвала такую конструкцию ASCIIZ-строкой (Zero – нуль). Например, чтобы перейти от корневого справочника файлов к подчиненному справочнику с именем SALES, надо указать в сегменте данных оператор

```
SALES_DIR  DB   \SALES',0
```

а в сегменте команд – операторы

```
       LEA   DX,SALES_DIR      ;Указать на имя справочника файлов
       MOV   AH,3BH            ;Перейти к этому справочнику
       INT   21H
       JNC   OKAY              ;Успешно?
       ...                     ;Нет. Считать код ошибки из AX
       ...
OKAY:
```

ФУНКЦИИ РАСШИРЕННОГО УПРАВЛЕНИЯ ФАЙЛАМИ

Подобно функциям для работы со справочниками файлов для большинства функций расширенного управления файлами требуется в качестве параметра ASCIIZ-строка – в данном случае строка, идентифицирующая файл. Подобная строка содержит имя дисковода и путь к файлу (если необходимо), после которых указывается имя файла, расширение имени (если оно есть) и нулевой байт. Например, для файла SALESQ4.NE, описание которого содержится в справочнике SALES, в сегменте данных надо указать оператор вида

```
SALESQ4$ DB '\SALES\SALESQ4.NE',0
```

Функции расширенного управления файлами используют два понятия, которые нам еще не встречались: *атрибут* файла и его *логический номер*. В операционной системе DOS версии 2 атрибут файла служит для его классификации. Каждая запись справочника файлов содержит *байт атрибута*, по битам которого можно

определить, описывает ли эта запись файл, доступный только для чтения (бит 0 равен 1), скрытый файл (бит 1 равен 1), системный файл (бит 2 равен 1), метку тома (бит 3 равен 1), подчиненный справочник файлов (бит 4 равен 1) или обычный файл (бит 5). Для файлов на жестком диске бит 5 представляет собой признак "архивирования": операционная система DOS делает его равным 1, если Вы изменяете этот файл, и обнуляет его, если Вы скопировали файл на гибкий диск с помощью команды BACKUP. Для файлов на гибком диске бит 5 всегда равен 1.

Функция 43 позволяет прочитать или изменить атрибут файла. Обычно не требуется изменять атрибут метки тома или подчиненного справочника файлов, но может понадобиться, например, сделать системный файл скрытым. (Под скрытым мы понимаем файл, имя которого не входит в распечатку справочника файлов, выдаваемую обычной командой DIR). Однако важнее всего то, что с помощью функции 43 можно сделать файл доступным *только для чтения*. Такие файлы *защищены от записи*; они не могут быть изменены или удалены.

При работе с операционной системой DOS версии 2 защитить файл от записи можно только с помощью функции 43! (В DOS версии 3 можно защитить файл от записи командой ATTRIB, но в DOS версии 2 такой команды нет.) В конце настоящего раздела, в подразделе "Файлы, защищенные от записи", мы рассмотрим программы, обеспечивающие защиту файлов от записи и снятие этой защиты.

Логический номер файла представляет собой число, идентифицирующее открытый файл или устройство ввода-вывода. Он похож на тот номер, который присваивается файлу оператором языка Бейсик вида OPEN "DATA" FOR OUTPUT AS#1. При вызове с диска операционная система DOS версии 2 выполняет следующие присваивания логических номеров:

Логический номер 0 отвечает стандартному устройству ввода. Операционная система DOS присваивает номер 0 клавиатуре, но Вы можете переприсвоить или "перенаправить" его какому-либо другому устройству (см. раздел "Переприсваивание стандартного входного и стандартного выходного устройства" в главе 1 руководства по операционной системе DOS). Например, Вам может понадобиться переприсвоить номер 0 удаленному терминалу.

Логический номер 1 отвечает стандартному устройству вывода. Операционная система DOS присваивает номер 1 экрану дисплея, но Вы можете переприсвоить его (см. предыдущую ссылку).

Логический номер 2 отвечает стандартному устройству вывода сообщений об ошибках. Он не может быть переприсвоен.

Логический номер 3 отвечает стандартному вспомогательному устройству.

Логический номер 4 отвечает стандартному принтеру.

Операционная система DOS сконфигурирована таким образом, что может воспринять еще четыре дополнительных определяемых пользователем логических номера, но при желании их число можно увеличить.

Например, можно воспользоваться функцией 40 для изображения сообщения на экране, т. е. на стандартном устройстве вывода. Для этого Ваш сегмент данных должен содержать текст сообщения, оформленный следующим образом:

```
MSG  DB "Пожалуйста, попробуйте еще раз"    ;Сообщение
     DB 13,10              ;Возврат каретки и переход к новой строке
MSGL EQU $-MSG             ;Длина сообщения
```

А сегмент команд должен содержать следующие команды, обеспечивающие выдачу сообщения:

```
LEA DX,MSG      ;Поместить смещение адреса сообщения в регистр DX
MOV CX,MSGL     ;Регистр CX служит счетчиком изображаемых символов
MOV BX,1        ;Загрузить логический номер экрана
MOV AH,40H      ;Задать номер функции
INT 21H         ;Изобразить сообщение
```

Обратите внимание на то, что в конце строки нет символа $, который требовался при вызове функции 9 (см. подраздел "Функции для работы с клавиатурой"); вместо этого мы использовали оператор MSGL EQU $-MSG для вычисления длины сообщения.

ПРОГРАММА ВЫДАЧИ СООБЩЕНИЙ ОБ ОШИБКАХ ОПЕРАЦИОННОЙ СИСТЕМЫ DOS ВЕРСИИ 2

Как уже упоминалось ранее, при обнаружении ошибки большинство новых функций операционной системы DOS версии 2 делают флаг переноса CF равным 1 и загружают код ошибки в регистр AX. Возможные ошибки перечислены в табл. 6.6, но чтобы Вы не тратили время на постоянное обращение к ней, в примере 6.4 показана процедура с именем SHOW_ERR, которая по ходу ошибки в регистре Ax изображает текст с ее описанием.

Если имя сегмента данных вызывающей программы не совпадает с именем сегмента данных процедуры SHOW_ERR, то не забудьте сохранить значение регистра DS вызывающей программы. Для этого в начале сегмента команд процедуры SHOW_ERR вставьте группу команд следующего вида:

```
PUSH DS         ;Сохранить значение DS вызывающей программы
MOV SI,DSEG     ;Инициировать DS
MOV DS,SI
```

Конечно, в конце сегмента команд надо вставить команду POP DS.

ПРИМЕР 6.4. ПРОГРАММА ВЫДАЧИ СООБЩЕНИЙ ОБ ОШИБКАХ ОПЕРАЦИОННОЙ СИСТЕМЫ DOS ВЕРСИИ 2

```
; Изобразить сообщение в зависимости от кода ошибки в регистре AX
; Значения регистров не изменяются
;
; Поместите следующие значения в сегмент данных
CR    EQU  13
LF    EQU  10
EOM   EQU  '$'
OUT_OF_RANGE  DB  'Код ошибки вне диапазона 1-18
              DB  CR,LF,EOM
ER1   DB  'Ошибка в номере функции ,CR,LF,EOM
ER2   DB  'Файл не найден',CR,LF,EOM
ER3   DB  'Путь к файлу не найден',CR,LF,EOM
ER4   DB  'Слишком много открытых файлов (не осталось логических номеров)
      DB  CR,LF,EOM
ER5   DB  'Доступ не разрешен',CR,LF,EOM
ER6   DB  'Ошибка в логическом номере',CR,LF,EOM
ER7   DB  'Блоки управления памятью разрушены',CR,LF,EOM
ER8   DB  'Недостаточно памяти',CR,LF,EOM
ER9   DB  'Ошибка в адресе блока памяти',CR,LF,EOM
ER10  DB  'Ошибочная операционная среда',CR,LF,EOM
ER11  DB  'Ошибка в формате',CR,LF,EOM
ER12  DB  'Ошибка в коде доступа',CR,LF,EOM
ER13  DB  'Ошибочные данные',CR,LF,EOM
ER14  DB  'Такого сообщения нет',CR,LF,EOM
ER15  DB  'Ошибка в указании дисковода',CR,LF,EOM
ER16  DB  'Попытка удалить текущий справочник файлов ,CR,LF,EOM
```

```
ER17 DB  'Неподходящее устройство',CR,LF,EOM
ER18 DB  'Файлов больше нет',CR,LF,EOM
ERTAB   DW   ER1,ER2,ER3,ER4,ER5,ER6,ER7,ER8,ER9
        DW   ER10,ER11,ER12,ER13,ER14,ER15,ER16,ER17,ER18
;
; Ниже следует основная процедура
;
SHOW_ERR   PROC
           PUSH  AX          ;Сохранить переданный код ошибки
           PUSH  BX          ;Сохранить другие рабочие регистры
           PUSH  CX
           CMP   AX,18       ;Убедиться, что код ошибки <= 18
           JG    O_O_R
           CMP   AX,0
           JG    IN_RANGE
O_O_R:     LEA   DX,OUT_OF_RANGE
           JMP   SHORT DISP_MSG
IN_RANGE:  LEA   BX,ERTAB-2  ;Указать на таблицу смещений
           SHL   AX,1        ;Указать на нужное смещение
           ADD   BX,AX
           MOV   DX,[BX]     ;Поместить адрес сообщения в DX
DISP_MSG:  MOV   AH,9
           INT   21H
           POP   DX
           POP   BX
           POP   AX
           RET
SHOW_ERR   ENDP
```

ФАЙЛЫ, ЗАЩИЩЕННЫЕ ОТ ЗАПИСИ

Как уже упоминалось в этом разделе, функция 43 операционной системы DOS версии 2 позволяет изменить атрибут файла. Кроме того, она позволяет защитить файлы от записи. После того как файл защищен от записи, пользователи могут читать его содержимое, но изменить или уничтожить его они не в состоянии. На попытку удалить защищенный от записи (или *доступный только для чтения*) файл операционная система DOS отреагирует сообщением "File not found" (Файл не найден).

В примере 6.5 показана программа LOCK, защищающая от записи заданный файл. Она похожа на операционную систему DOS тем, что позволяет Вам вводить имя дисковода и путь к файлу, если файл находится на другом диске или описан в другом справочнике файлов. Обратите внимание на то,что программа LOCK вызывает процедуру SHOW_ERR для выдачи сообщений об ошибках. В примере 6.6 показана почти идентичная ей программа UNLOCK, снимающая защиту от записи.

ПРИМЕР 6.5. ЗАЩИТА ФАЙЛА ОТ ЗАПИСИ

```
; Эта процедура защищает файл от записи с помощью установки бита
; "только-для-чтения" в байте атрибута этого файла
;
     EXTRN        SHOW_ERR:FAR
;
; Поместите следующие операторы в сегмент данных
;
PROMPT      DB    Какой файл Вы хотите защитить?
PROMPTL     DB    $-PROMPT
FILESPEC    DB    60 DUP(?)
;
; Ниже следует основная процедура
;
```

```
LOCK  PROC
      MOV   AH,40H            ;Изобразить приглашение к вводу
      MOV   BX,1
      MOV   CX,PROMPTL
      LEA   DX,PROMPT
      INT   21H
      MOV   AH,3FH            ;Получить от пользователя идентификатор файла
      MOV   BX,0
      MOV   CX,60
      LEA   DX,FILESPEC
      INT   21H
;
; Преобразовать ввод в ASCIIZ-строку с помощью замены CR на нуль
;
      SUB   AX,2              ;Указать на символ возврата каретки CR
      MOV   BX,AX             ;Поместить указатель в BX
      MOV   FILESPEC[BX],0    ;Заменить CR на 0
;
; Изменить атрибут файла на "только-для-чтения"
;
      MOV   AH,43H            ;Считать байт атрибута
      MOV   AL,0
      LEA   DX,FILESPEC
      INT   21H
      JNC   SET_ATT           ;Ошибка при чтении?
      CALL  SHOW_ERR          ; Если да, выдать сообщение об ошибке
      RET
SET_ATT: OR  CX,1             ; В противном случае установить бит
;                                "только-для-чтения"
      MOV   AL,1              ;Записать байт атрибута
      MOV   AH,43H
      LEA   DX,FILESPEC
      INT   21h
      JNC   LEAVE             ;Ошибка при записи?
      CALL  SHOW_ERR          ; Если да, выдать сообщение об ошибке
LEAVE: RET
LOCK  ENDP
```

ПРИМЕР 6.6. СНЯТИЕ С ФАЙЛА ЗАЩИТЫ ОТ ЗАПИСИ

```
; Эта процедура снимает защиту файла от записи с помощью обнуления
; бита "только-для-чтения" в байте атрибута этого файла
;
      EXTRN       SHOW_ERR:FAR
;
; Поместите следующие операторы в сегмент данных
;
PROMPT     DB    С какого файла Вы хотите снять защиту?
PROMPTL    DB    $-PROMPT
FILESPEC   DB    60 DUP(?)
;
; Ниже следует основная процедура
;
UNLOCK  PROC
      MOV   AH,40H            ;Изобразить приглашение к вводу
      MOV   BX,1
      MOV   CX,PROMPTL
      LEA   DX,PROMPT
      INT   21H
      MOV   AH,3FH            ;Получить от пользователя идентификатор файла
      MOV   BX,0
      MOV   CX,60
      LEA   DX,FILESPEC
      INT   21H
;
; Преобразовать ввод в ASCIIZ-строку с помощью замены CR на нуль
;
```

```
       SUB   AX,2               ;Указать на символ возврата каретки CR
       MOV   BX,AX              ;Поместить указатель в BX
       MOV   FILESPEC[BX],0     ;Заменить CR на 0
;
; Изменить атрибут файла с "только-для-чтения" на "чтение/запись"
;
       MOV   AH,43H             ;Считать байт атрибута
       MOV   AL,0
       LEA   DX,FILESPEC
       INT   21H
       JNC   CLR_ATT            ;Ошибка при чтении?
       CALL  SHOW_ERR           ; Если да, выдать сообщение об ошибке
       RET
CLR_ATT: AND  CX,0FEH           ; В противном случае обнулить бит
;                                  "только-для-чтения"
       MOV   AL,1               ;Записать байт атрибута
       MOV   AH,43H
       LEA   DX,FILESPEC
       INT   21H
       JNC   LEAVE              ;Ошибка при записи?
       CALL  SHOW_ERR           ; Если да, выдать сообщение об ошибке
LEAVE: RET
UNLOCK   ENDP
```

6.4. РАБОТА С КЛАВИАТУРОЙ

В этом разделе вкратце (и упрощенно) излагаются принципы работы клавиатуры и возможные способы взаимодействия с ней.

СИСТЕМА ASCII

Клавиатуры многих микропроцессорных систем соединяются с ЭВМ посредством микросхемы, называемой *шифратором*, которая преобразует каждый удар по клавише в 8-битовый ASCII-код. Система ASCII представляет собой набор числовых кодов, используемых ЭВМ для обмена данными. Как показано в табл. 6.7, набор символов ASCII представлен 128 кодами (от 00H до 7FH). Чтобы отыскать ASCII-код данного символа, надо к старшей цифре, указанной в верхней строке, приписать младшую цифру из крайней левой колонки. Например, латинской букве A соответствует старшая цифра 4, а младшая – 1, так что кодом этой буквы служит 41H.

Обратите внимание, что наряду с обычными буквами, цифрами и знаками набор символов ASCII содержит управляющие символы, например символ возврата каретки CR (Carriage Return), прогона страницы FF (Form Feed), перехода на следующую строку LF (Line Feed). К ним принадлежат некоторые символы, используемые в коммуникационных протоколах, например символ подтверждения приема ACK (Acknowledge), начала текста STX (Start of Text) и конца передачи EOT (End of Transmission). Однако если Вы тщательно изучите табл. 6.7 (откровенно говоря, это не слишком увлекательное занятие), то увидите, что в нее не включены специальные клавиши IBM PC. Например, Вы не найдете в табл. 6.7 кодов функциональных клавиш или клавиши фиксирования цифрового регистра. Идентификация таких клавиш представляла особую задачу для разработчиков клавиатуры фирмы IBM, и они нашли довольно интересное решение, на котором стоит остановиться подробнее.

Таблица 6.7. **Стандартный набор символов ASCII**

	Старшие разряды	0	1	2	3	4	5	6	7
Младшие разряды		000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P		p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	,	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	—	=	M	]	m	}
E	1110	SO	RS	.	>	N		n	~
F	1111	SI	US	/	?	O	—	o	DEL

Примечание.

NUL — пустой символ (null)

SOH — начало заголовка (start of heading)

STX — начало текста (start of text)

ETX — конец текста (end of text)

EOT — конец передачи (end of transmission)

ENQ — запрос подтверждения (enquiry)

ACK — подтверждение (acknowledge)

BEL — звуковой сигнал (bell)

BS — возврат на одну позицию (backspace)

HT — горизонтальная табуляция (horizontal tabulation)

LF — переход к новой строке (line feed)

VT — вертикальная табуляция (vertical tabulation)

FF — переход к новой странице (form feed)

CR — возврат каретки (carriage return)

SO — нижний регистр (shift out)

SI — верхний регистр (shift in)

DL — завершение сеанса связи (data link escape)

DC — управление устройством (device control)

NAK — ошибка передачи (negative acknowledge)

SYN — холостые данные синхронной передачи (synchronous idle)

ETB — конец передаваемого блока (end of transmission block)

CAN — отмена (cancel)

EM — конец носителя данных (end of medium)

SUB — подстановка (substitute)

ESC — прекращение (escape)

FS — разделитель файлов (file separator)

GS — разделитель групп (group separator)

RS — разделитель записей (record separator)

US — разделитель элементов (unit separator)

SP — пробел (space)

DEL — удаление (delete)

ПРИНЦИП ДЕЙСТВИЯ КЛАВИАТУРЫ IBM PC

Клавиатура IBM PC работает следующим образом: всякий раз, когда Вы нажимаете или отпускаете клавишу, встроенное в клавиатуру устройство запоминает в своей внутренней памяти один байт. Этот байт содержит код *нажатия* или *отпускания* (соответственно 1 и 0) в седьмом бите и идентификатор клавиши,

или scan-код, в остальных битах. Всего возможно 83 scan-кода, по одному для каждой клавиши клавиатуры (рис. 6.5). После того как встроенное в клавиатуру устройство запомнило какие-либо данные в своей памяти, оно посылает системе BIOS IBM PC прерывание типа 9, сообщающее, что у него есть данные для передачи.

Память клавиатуры способна хранить до 20 байтов, или до 10 операций нажатия и отпускания (т.е. ударов по клавишам). Это позволяет Вам набирать текст даже в то время, когда микропроцессор ЭВМ занят, например выдает содержимое экрана на принтер.

Если прерывания разрешены, то процедура обработки прерывания типа 9 KB_INT, входящая в состав системы BIOS, считает байты из памяти клавиатуры и преобразует их в коды символов. Частью процесса преобразования является проверка того, не было ли некоторых нажатий без последующего отпускания; это позволяет формировать коды символов, отражающие удерживание в нажатом состоянии регистровых клавиш Shift, Ctrl или Alt во время нажатия на другую клавишу. По завершении работы процедура обработки прерывания типа 9 запоминает scan-код и код символа в *буфере клавиатуры*, находящемся в памяти ЭВМ.

Этот буфер рассчитан на сохранение результатов 15 ударов по клавишам; при таком размере буфера скорость работы программного обеспечения позволяет успевать за самыми быстрыми машинистками. Но если Вам вдруг удастся нажать на клавишу в то время, когда буфер полон, то система BIOS проигнорирует эту клавишу и выдаст звуковой сигнал.

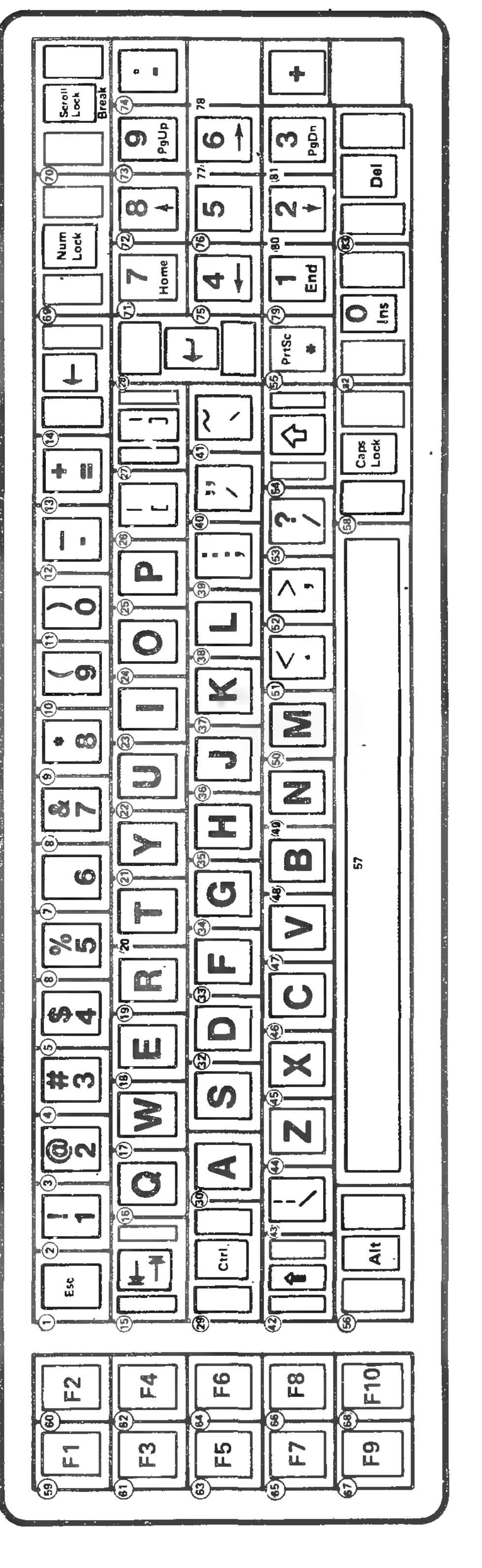


Рис. 6.5. Клавиатура и scan-коды

РАСШИРЕННАЯ СИСТЕМА ASCII

Ранее уже упоминалось, что клавиатура IBM PC имеет так много клавиш и настолько отличается от стандартной клавиатуры ЭВМ, что для представления генерируемых ею кодов символов не хватает стандартного набора из 128 ASCII-кодов. Всего в IBM PC предусмотрено 256 кодов, называемых фирмой IBM расширенной системой ASCII.

КОДЫ СИМВОЛОВ И SCAN-КОДЫ

В табл. 6.8 и 6.9 приведены scan-коды и символы, которые могут быть получены с помощью 83 клавиш. Они расположены в двух таблицах потому, что клавиши цифровой клавиатуры генерируют разные коды в зависимости от положения клавиши цифрового регистра NumLock.

В табл. 6.8 показаны символы, генерируемые клавишами 1 – 70 в нижнем (основном) и верхнем регистрах, а также при нажатых клавишах управляющего регистра Ctrl и альтернативного регистра Alt. За исключением комбинаций клавиш, в которые входит Alt, большинство отдельных клавиш и их комбинаций генерируют стандартный ASCII-код.

Клавиши, не имеющие эквивалента в системе ASCII (например, Alt и функциональные клавиши) возвращают 0 в качестве кода символа. В этом случае Ваша программа должна проверить второй код, чтобы определить, какая клавиша или какие клавиши были нажаты. Кратко мы обсудим эти так называемые расширенные коды. Другие ("не ASCII") клавиши вызывают выполнение специальных операций, например выдачу на принтер содержимого экрана. Мы рассмотрим их в подразделе "Специальные комбинации клавиш". Наконец, есть некоторые комбинации клавиш, которые система BIOS просто игнорирует; в табл. 6.9 они помечены н/в.

РАСШИРЕННЫЕ КОДЫ

Когда Вы запрашиваете чтение символа с клавиатуры, система BIOS возвращает ASCII-код этого символа и его scan-код в регистрах AL и AH соответственно. Другие ("не ASCII") клавиши возвращают 0 в регистре AL и расширенный код в регистре AH. В табл. 6.10 перечислены расширенные коды и показаны комбинации клавиш, которые им соответствуют. Обратите внимание на то, что расширенные коды в интервале 3 – 53 соответствуют scan-кодам.

СПЕЦИАЛЬНЫЕ КОМБИНАЦИИ КЛАВИШ

Следующие комбинации клавиш вызывают специальные действия:

Alt-Ctrl-Del вызывает рестарт или "перевызов" операционной системы ЭВМ.

Ctrl-Break инициирует прерывание типа 1B (клавиша прерывания). Эта комбинация клавиш возвращает 0 в регистрах AL и AH.

Ctrl-NumLock заставляет ЭВМ ждать, пока Вы не нажмете какую-либо клавишу, отличную от клавиши NumLock. Это даст Вам возможность приостановить операцию (например, распечатку программы), а затем возобновить ее исполнение.

Таблица 6.8. **Символы, генерируемые клавишами 1 — 70**

Scan-код DEC	HEX	Основной регистр	Верхний регистр	CTRL	ALT
1	1	ESC	ESC	ESC	н/в
2	2	1	!	н/в	Прим. 1
3	3	2	@	NUL (прим. 1)	Прим. 1
4	4	3	#	н/в	Прим. 1
5	5	4	$	н/в	Прим. 1
6	6	5	%	н/в	Прим. 1
7	7	6	^	RS	Прим. 1
8	8	7	&	н/в	Прим. 1
9	9	8	*	н/в	Прим. 1
10	A	9	(	н/в	Прим. 1
11	B	0	)	н/в	Прим. 1
12	C	–	_	US	Прим. 1
13	D	=	+	н/в	Прим. 1
14	E	Пробел	Пробел	DEL	н/в
15	F	→|	|← (прим. 1)	н/в	н/в
16	10	q	Q	DC1	Прим. 1
17	11	w	W	ETB	Прим. 1
18	12	e	E	ENQ	Прим. 1
19	13	r	R	DC2	Прим. 1
20	14	t	T	DC4	Прим. 1
21	15	y	Y	EM	Прим. 1
22	16	u	U	NAK	Прим. 1
23	17	i	I	HT	Прим. 1
24	18	o	O	SI	Прим. 1
25	19	p	P	DLE	Прим. 1
26	1A	[	{	ESC	н/в
27	1B	]	}	GS	н/в
28	1C	CR	CR	LF	н/в
29	1D CTRL	н/в	н/в	н/в	н/в
30	1E	a	A	SOH	Прим. 1
31	1F	s	S	DC3	Прим. 1
32	20	d	D	EOT	Прим. 1
33	21	f	F	ACK	Прим. 1
34	22	g	G	BEL	Прим. 1
35	23	h	H	BS	Прим. 1
36	24	j	J	LF	Прим. 1
37	25	k	K	VT	Прим. 1
38	26	l	L	FF	Прим. 1
39	27	;	:	н/в	н/в
40	28	'	"	н/в	н/в
41	29	`	~	н/в	н/в

Окончание табл. 6.8

Scan-код DEC	HEX	Основной регистр	Верхний регистр	CTRL	ALT
42	2A SHIFT	н/в	н/в	н/в	н/в
43	2B	\	|	FS	н/в
44	2C	z	Z	SUB	Прим. 1
45	2D	x	X	CAN	Прим. 1
46	2E	c	C	ETX	Прим. 1
47	2F	v	V	SYN	Прим. 1
48	30	b	B	STX	Прим. 1
49	31	n	N	SO	Прим. 1
50	32	m	M	CR	Прим. 1
51	33	.	<	н/в	н/в
52	34	,	>	н/в	н/в
53	35	/	?	н/в	н/в
54	36 SHIFT	н/в	н/в	н/в	н/в
55	37	*	Прим. 2	Прим. 1	н/в
56	38 ALT	н/в	н/в	н/в	н/в
57	39 SP	SP	SP	SP	SP
58	3A CAPS LOCK	н/в	н/в	н/в	н/в
59	3B F1	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)
60	3C F2	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)
61	3D F3	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)
62	3E F4	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)
63	3F F5	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)
64	40 F6	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)
65	41 F7	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)
66	42 F8	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)
67	43 F9	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)
68	44 F10	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)	NUL (прим. 1)
69	45 NUM LOCK	н/в	н/в	Пауза (прим. 2)	н/в
70	46 SCROLL LOCK	н/в	н/в	Прерывание (прим. 2)	н/в

Примечания: DEC — десятичная цифра;
HEX — шестнадцатеричное значение.
Прим. 1 означает см. подраздел "Расширенные коды";
прим. 2 означает см. подраздел "Специальные комбинации клавиш";
н/в означает — не влияет.

Shift-PrtSc инициирует прерывание типа 5 (печать содержимого экрана).
Процедура системы BIOS, обслуживающая клавиатуру, воспринимает клавиши Ctrl, Shift, NumLock, ScrollLock и Ins как управляющие.
Программа обработки прерывания типа 16 (обмен данными с клавиатурой) возвращает байт состояния регистровых клавиш, который сообщает о том, какие из этих клавиш были нажаты.

Таблица 6.9. **Символы, генерируемые клавишами 71 — 83 (цифровая клавиатура)**

Scan-код DEC	Scan-код HEX	NUM LOCK	Основной регистр	ALT	CTRL
71	47	7	Home — начало экрана (прим. 1)	н/в	Очистка экрана
72	48	8	↓ (прим. 1)	н/в	н/в
73	49	9	PgUp — предыдущая страница (прим. 1)	н/в	Начало текста и начало экрана
74	4A	–	–	н/в	н/в
75	4B	4	← (прим. 1)	н/в	Предыдущее слово (прим. 1)
76	4C	5	н/в	н/в	н/в
77	4D	6	→ (прим. 1)	н/в	Следующее слово (прим. 1)
78	4E	+	+	н/в	н/в
79	4F	1	End — конец экрана (прим. 1)	н/в	Очистка до конца строки (прим. 1)
80	50	2	↓ (прим. 1)	н/в	н/в
81	51	3	PgDn-следующая страница (прим. 1)	н/в	Очистка до конца экрана (прим. 1)
82	52	0	Ins — вставить	н/в	н/в
83	53		Del — удалить (прим. 1, 2)	Прим. 2	Прим. 2

Примечания. DEC — десятичная цифра;
HEX — шестнадцатеричное значение.
Прим. 1 — см. подраздел "Расширенные коды";
прим. 2 — см. подраздел "Специальные комбинации клавиш";
н/в — не влияет.

Таблица 6.10. **Расширенные коды клавиатуры**

Расширенный код	Функция
3	Ctrl @
F	←
10 – 19	ALT Q, W, E, R, T, Y, U, I, O, P (второй ряд)
1E – 26	ALT A, S, D, F, G, H, J, K, L (третий ряд)
2C – 32	ALT Z, X, C, V, B, N, M (четвертый ряд)
3B – 44	Функциональные клавиши F1 – F10 (основной регистр)
47	Home — начало экрана
48	↑

Расширенный код	Функция
49	PgUp и Home — предыдущая страница и начало экрана
4B	←
4D	→
4K	End — конец экрана
50	↓
51	PgDn и Home — следующая страница и начало экрана
52	Ins — вставить
53	Del — удалить
54 — 5D	Shift F1 — F10
5E — 67	Ctrl F1 — F10
68 — 71	Alt F1 — F10
72	Ctrl PrtSc — дублировать изображенный текст на принтере/не дублировать
73	Ctrl ← — предыдущее слово
74	Ctrl → — следующее слово
75	Ctrl End (очистка до конца строки)
76	Ctrl PgDn (очистка до конца экрана)
77	Ctrl Home (очистка до конца экрана и переход к началу экрана)
78 — 83	Alt 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, –, = (верхний ряд)
84	Ctrl PgUp (прокрутка 25 строк текста и переход к началу экрана)

ПРЕРЫВАНИЯ ДЛЯ РАБОТЫ С КЛАВИАТУРОЙ

Ранее в этой главе уже упоминались два типа прерываний, предназначенных для обмена данными с клавиатурой: прерывание типа 16 (обмен данными с клавиатурой), которое обрабатывается системой BIOS, и прерывание типа 21, с помощью которого можно вызвать функции для работы с клавиатурой, встроенные в операционную систему DOS. Обсудим каждое прерывание и выясним, как их надо использовать.

ОПЕРАЦИИ, ОБЕСПЕЧИВАЕМЫЕ ПРЕРЫВАНИЕМ ТИПА 16

Прерывание типа 16 (обмен данными с клавиатурой) предусматривает весьма примитивные операции, которые могут быть полезны для написания программы, исполняемой независимо от операционной системы DOS. В зависимости от значения регистра AH прерывание типа 16 позволяет выбрать одну из следующих трех операций:

Если (AH) = 0, то программа обработки прерывания типа 16 (KEYBOARD_IO) считывает из клавиатурного буфера scan-код очередной клавиши в регистр AH и код ее символа в регистр AL, а затем продвигает указатель буфера. Если буфер пуст, то программа KEYBOARD_IO ожидает нажатия клавиши и затем продолжает его обработку.

Если (AH) = 1, то программа KEYBOARD_IO возвращает информацию о состоянии буфера клавиатуры во флаге нуля ZF. Если буфер пуст, то ZF равен 1. Если

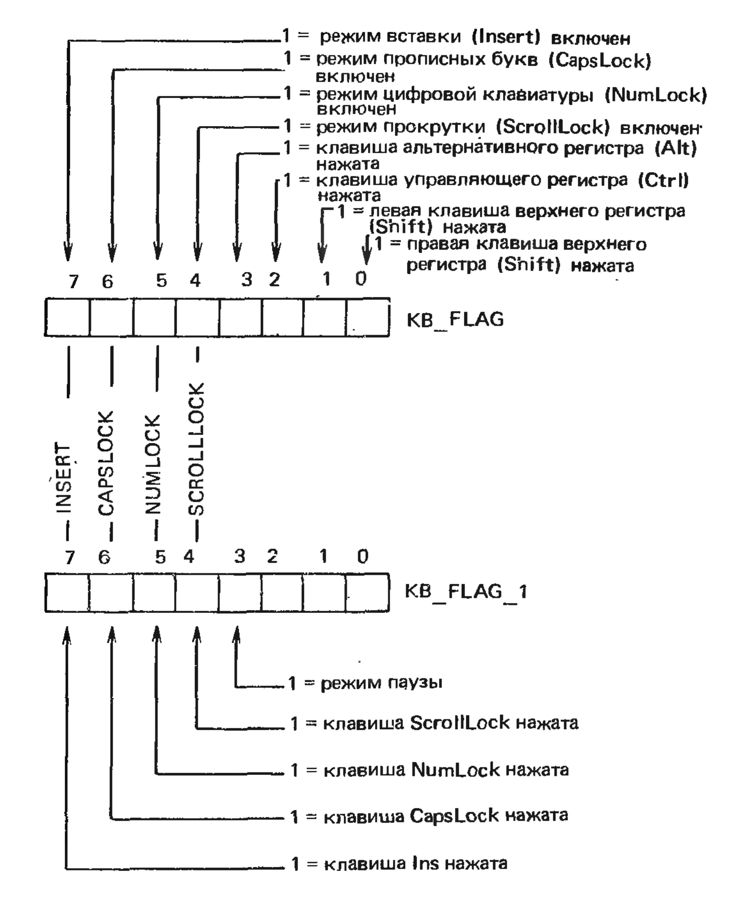


Рис. 6.6. Байты состояния регистровых клавиш

для считывания имеется хотя бы один код клавиши, то ZF равен 0. Это означает, что очередной символ находится в регистре AX, а остальные символы – в буфере.

Если (AH) = 2, то программа KEYBOARD_IO возвращает в регистре AL байт состояния клавиатуры. Описание этого байта приводится ниже.

Программа KEYBOARD_IO воздействует только на регистр AX и флаги.

На верхней половине рис. 6.6 показаны значения отдельных битов байта состояния, возвращаемого при (AH) = 2. Старшие четыре бита этого байта (байта KB_FLAG в листинге системы BIOS) показывают, какие из различных режимов клавиатуры включены (1) или выключены (0), а младшие четыре бита показывают, удерживаются ли клавиши Alt, Ctrl или Shift в нажатом состоянии.

На нижней половине рис. 6.6 показан байт KB_FLAG_1, расположенный рядом с байтом KB_FLAG. Он предоставляет дополнительную информацию о состоянии клавиатуры. Программа KEYBOARD_IO пользуется им для внутренних целей и не дает возможности получить его через регистр. Однако байт KB_FLAG_1 расположен в памяти непосредственно за байтом KB_FLAG, так что для его получения можно считать содержимое ячейки 418H (байт KB_FLAG имеет адрес 417H).

Как уже упоминалось, прерывание типа 16 принадлежит к числу тех, которые позволяют создавать программы, не зависящие от операционной системы DOS. Большинству пользователей целесообразнее пользоваться более гибкими возможностями функций для работы с клавиатурой, вызываемых с помощью прерывания типа 21.

ВЫЗОВЫ ФУНКЦИЙ ДЛЯ РАБОТЫ С КЛАВИАТУРОЙ ЧЕРЕЗ ПРЕРЫВАНИЕ ТИПА 21

В разд. 6.3 мы обсуждали функции для обмена данными с клавиатурой, вызываемые посредством прерывания типа 21. Они включали четыре функции чтения отдельного кода клавиши (1, 6, 7 и 9), функцию чтения строки (А), функцию для проверки наличия в буфере клавиатуры хотя бы одного кода (В) и функцию, опустошающую буфер клавиатуры и вызывающую другую функцию (С). В том разделе мы в качестве примеров демонстрировали короткую программу, ожидавшую нажатия на клавиши Д или Н, и процедуру READ_KEYS, обеспечивающую чтение строки с клавиатуры. Однако мы не рассмотрели клавиши, выдающие так называемый расширенный код.

Если Вы помните, система BIOS преобразует нажатие на клавишу в scan-код и код символа в регистрах АН и AL соответственно. Некоторые клавиши, например функциональные клавиши или клавиши на цифровой клавиатуре, возвращают нуль в качестве кода символа. Это означает, что пользователь набрал одну из комбинаций клавиш, перечисленных в табл. 6.10; в этом случае для получения кода символа надо выполнить еще одну операцию считывания кода клавиши.

Предположим, например,что Ваша программа изображает меню и предлагает пользователю сделать выбор нажатием клавиши F1, F2 или F3. Программа должна игнорировать нажатие любых других клавиш. Для обработки выбора пользователя надо использовать последовательность команд следующего вида:

```
KEY:    MOV  AH,8      ;Ждать нажатия клавиши (не изображая ее символ)
        INT  21H
        CMP  AL,0      ;Расширенный код?
        JNZ  ERROR     ; Если нет, выдать сообщение об ошибке
        MOV  AH,8      ; В противном случае считать расширенный код
        INT  21H
        CMP  AL,3BH    ;Нажата клавиша F1?
        JE   F1
        CMP  AL,3CH    ;Нажата клавиша F2?
        JE   F2
        CMP  AL,3DH    ;Нажата клавиша F3?
        JE   F3
ERROR:  ... (Выдать на экран "Нажмите, пожалуйста, F1, или F2, или F3")
        ...
        JMP  SHORT KEY
```

6.5. ПРЕОБРАЗОВАНИЕ ЧИСЕЛ ИЗ ASCII-КОДОВ В ДВОИЧНУЮ СИСТЕМУ

Как Вы теперь знаете, система BIOS преобразует символы, которые набираются на клавиатуре, в ASCII-коды. Если эти символы образуют *числа*, то процессор может выполнять над ними арифметические операции лишь после преобразования в двоичную или двоично-десятичную систему. И, наоборот, перед изображением числа на экране или выдачей на принтер его надо преобразовать в ASCII-коды.

В этом разделе мы рассмотрим обе задачи: преобразование чисел из ASCII-кодов в двоичный код и преобразование двоичного числа в ASCII-коды. (Преобразо-

Таблица 6.11. **ASCII-коды десятичных цифр**

ASCII-код [1]	30	31	32	33	34	35	36	37	38	39
Десятичная цифра	0	1	2	3	4	5	6	7	8	9

[1] Значения приведены в шестнадцатеричной системе.

вание из ASCII-кодов в BCD-числа и преобразование BCD-чисел в ASCII-коды осуществляется аналогично, но, как говорится в учебниках, "решение этой задачи предоставляется читателю".)

ПРЕОБРАЗОВАНИЕ СТРОКИ ASCII-КОДОВ В ДВОИЧНОЕ ЧИСЛО

В табл. 6.11 показаны ASCII-коды десятичных цифр от 0 до 9. Как видите, нас могут интересовать значения только тех ASCII-кодов, которые находятся в интервале от 30H до 39H. Вы должны также учесть, что двоичный эквивалент десятичной цифры есть не что иное, как младшие четыре бита ее ASCII-кода.

Ранее уже упоминалось, что десятичные числа могут быть представлены в виде суммы цифр, умноженных на степени числа 10. Например

$$237 = (7 * 1) + (3 * 10) + (2 * 100)$$

или

$$237 = (7 * 10^0) + (3 * 10^1) + (2 * 10^2).$$

Так как цифры числа вводятся по одной, то программа преобразования из ASCII-кодов в двоичное число должна включать операцию умножения на 10. Например, если пользователь набирает 93, то перед сложением с 3 надо умножить 9 на 10. В общем случае порядок процесса преобразования следующий:

Программа преобразует первую (старшую) цифру в двоичное число обнулением четырех старших битов ее ASCII-кода. Это двоичное значение запоминается в качестве промежуточного результата.

Программа преобразует следующую цифру в двоичное число, затем умножает промежуточный результат на 10 и добавляет к полученному произведению значение цифры (модифицируя тем самым промежуточный результат).

АЛГОРИТМ ПРЕОБРАЗОВАНИЯ ASCII-КОДОВ В ДВОИЧНОЕ ЧИСЛО

Обычно требуется преобразовывать как положительные, так и отрицательные числа, а также числа с десятичной точкой. Поэтому наша программа преобразования должна учитывать возможность появления еще и символов минус (–) и точка (.). На рис. 6.7 показана блок-схема алгоритма преобразования находящейся в памяти строки ASCII-кодов в двоичное число со знаком (в дополнительном коде). Будем предполагать, что для его представления достаточно 16 битов, т. е. что оно лежит в интервале от –32768 до +32767.

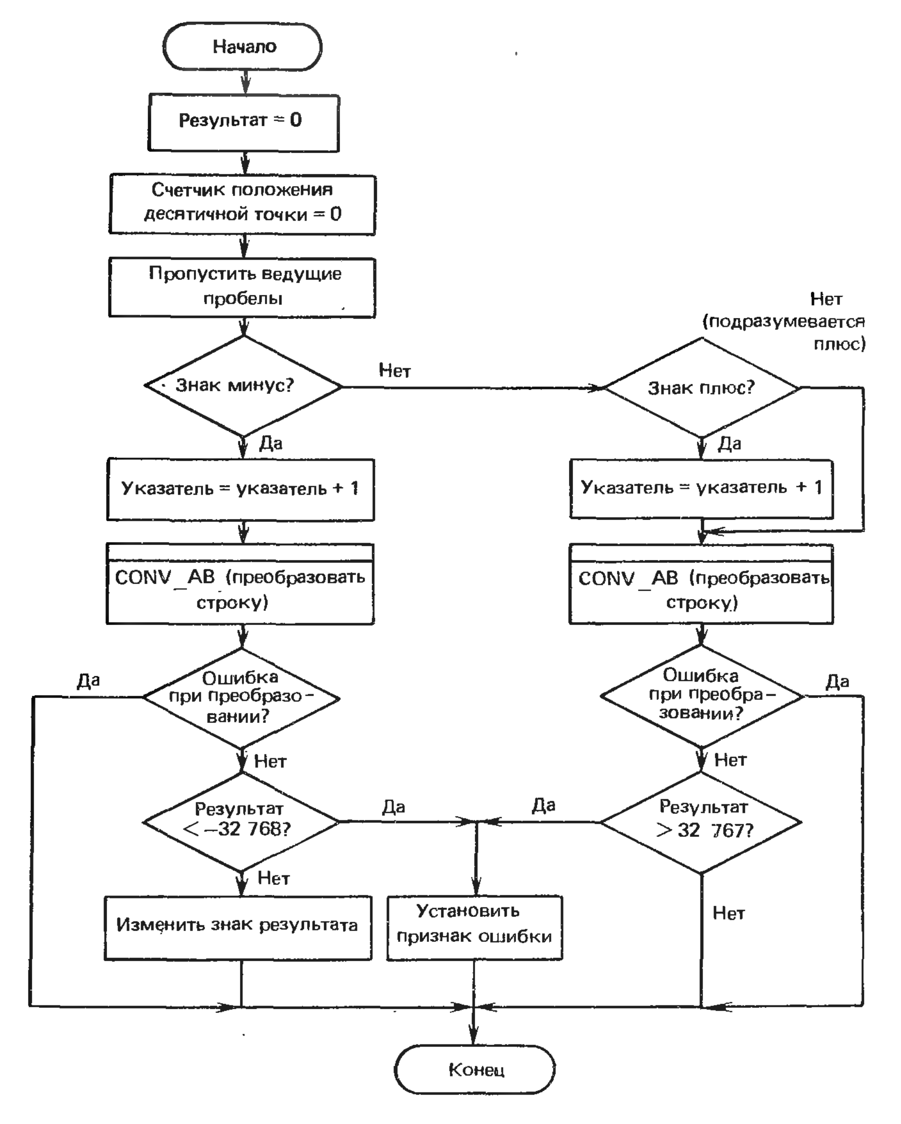


Рис. 6.7. Алгоритм преобразования строки ASCII-кодов в двоичное число

Вначале результат и счетчик положения десятичной точки (число цифр справа от десятичной точки) полагаются равными нулю. Затем программа пропускает все ведущие пробелы, после чего выбирает один из двух путей, связанных с преобразованием положительных и отрицательных чисел.

Оба пути почти идентичны, только преобразуемое отрицательное число надо сравнить с – 32768 и обратить, а положительное число сравнивается с 32767. Фактическое преобразование выполняется процедурой CONV_AB, блок-схема которой приведена на рис. 6.8.

Процедура CONV_AB начинает с проверки, является ли очередной символ строки десятичной точкой. Если это так, то процедура CONV_AB записывает число оставшихся символов в *счетчик положения десятичной точки*, а затем продвигает

указатель строки. Если очередной символ не является цифрой, то процедура CONV_AB объявляет его *ошибочным* и устанавливает признак ошибки, а затем возвращает управление основной программе.

При обнаружении *правильного* символа (т. е. цифры) процедура CONV_AB умножает текущий промежуточный результат на 10, затем преобразует ASCII-код цифры в число и добавляет его к полученному произведению. Если при сложении возникает перенос, то процедура CONV_AB устанавливает признак ошибки и воз-

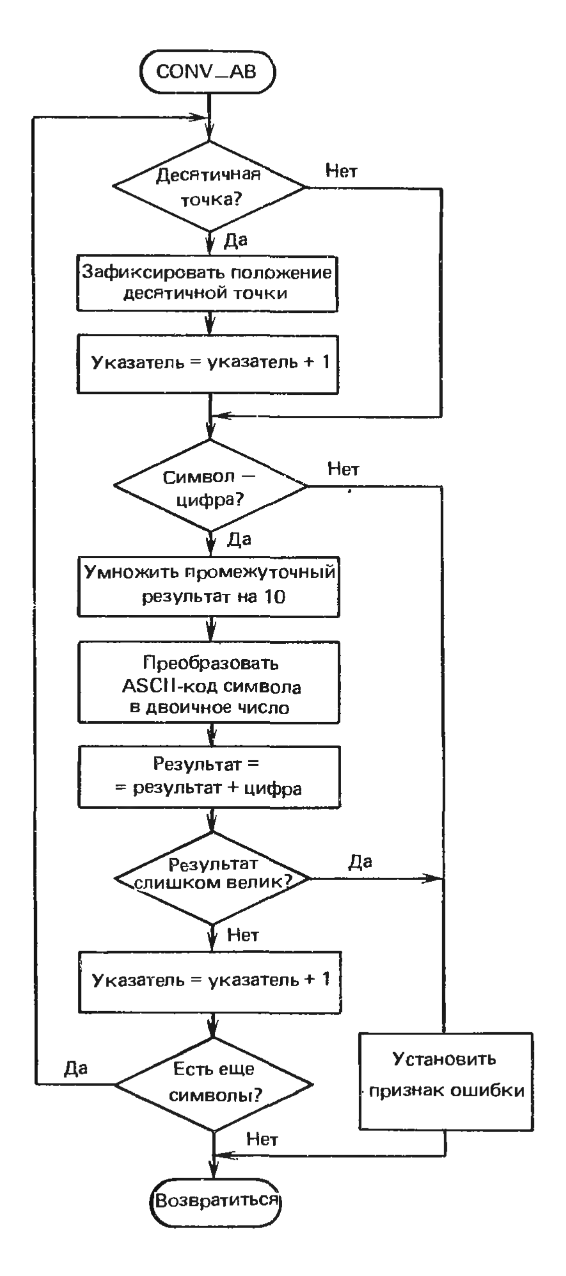


Рис. 6.8. Процедура, используемая в алгоритме преобразования рис. 6.7.

вращает управление основной программе. В противном случае она продвигает указатель и возвращается к команде проверки совпадения символа с десятичной точкой. Когда обработка всей строки завершена, процедура CONV_AB возвращает управление основной программе.

ПРОГРАММА ПРЕОБРАЗОВАНИЯ ASCII-КОДОВ В ДВОИЧНОЕ ЧИСЛО

В примере 6.7 показана процедура, реализующая описанный выше алгоритм. Эта процедура (ASCII_BIN) преобразует находящуюся в сегменте данных строку ASCII-кодов (скажем, введенную процедурой READ_KEYS из примера 6.2) в 16-битовое число со знаком.

Процедура ASCII_BIN берет начальный адрес строки из регистра BX, а счетчик символов (не более 7) из регистра CX. Она возвращает 16-битовое значение в регистре AX, а число цифр после десятичной точки (если таковая есть) в регистре DX. Адрес первого *ошибочного* символа возвращается в регистр DI.

ПРИМЕР 6.7. ПРЕОБРАЗОВАНИЕ ДЕСЯТИЧНОГО ЧИСЛА, ЗАПИСАННОГО В ASCII-КОДАХ, В ДВОИЧНОЕ ЧИСЛО

```
; Эта процедура преобразует строку ASCII-символов, находящуюся в сег-
; менте данных, в ее 16-битовый двоичный эквивалент в обратном коде
; Начальный адрес строки берется из регистра BX, а счетчик символов -
; из регистра CX
; По возвращению 16-битовое значение находится в регистре AX, а счет-
; чик числа цифр после десятичной точки - в регистре DX. Адрес первого
; не преобразуемого символа находится в регистре DI
; Если строка содержит более семи символов, или число лежит вне до-
; пустимого диапазона (больше 32 767 или меньше -32 768), или строка
; содержит не преобразуемый символ, то флаг переноса CF полагается
; равным 1. Если преобразование выполнено без ошибок, то CF равен 0
; и DI содержит 0FFH
; Содержимое регистров BX и CX не изменяется
;
ASCII_BIN   PROC
            PUSH  BX                     ;Сохранить значения BX и CX
            PUSH  CX
            SUB   AX,AX                  ;Начальные значения: результат = 0,
            SUB   DX,DX                  ; счетчик цифр после точки = 0,
            MOV   DI,0FFH                ; плохих символов нет
            CMP   CX,7                   ;Слишком длинная строка?
            JA    NO_GOOD                ; Да. Установить CF и выйти
BLANKS:     CMP   BYTE PTR [BX],         ;Пропустить ведущие пробелы
            JNE   CHK_NEG
            INC   BX
            LOOP  BLANKS
CHK_NEG:    CMP   BYTE PTR [BX],'-'      ;Отрицательное число?
            JNE   CHK_POS
            INC   BX                     ; Да. Продвинуть указатель,
            DEC   CX                     ;  уменьшить счетчик и
            CALL  CONV_AB                ;  преобразовать строку
            JC    THRU
            CMP   AX,32768               ;Число слишком мало?
            JA    NO_GOOD
            NEG   AX                     ; Нет. Обратить знак результата
            JS    GOOD
CHK_POS:    CMP   BYTE PTR [BX],'+'      ;Положительное число?
            JNE   GO_CONV
            INC   BX                     ; Да. Продвинуть указатель,
            DEC   CX                     ;  уменьшить счетчик и
GO_CONV:    CALL  CONV_AB                ;  преобразовать строку
            JC    THRU
```

```
            CMP     AX,32767            ;Число слишком велико?
            JA      NO_GOOD
GOOD:       CLC
            JNC     THRU
NO_GOOD     STC                         ; Да. Установить флаг переноса
THRU:       POP     CX                  ;Восстановить значения регистров
            POP     BX
            RET                         ; и выйти из процедуры
ASCII_BIN   ENDP
;
;Следующая ниже процедура выполняет собственно преобразование.
;
CONV_AB     PROC
            PUSH    BP                  ;Сохранить рабочие регистры
            PUSH    BX
            PUSH    SI
            MOV     BP,BX               ;Поместить указатель в BP
            SUB     BX,BX               ; и обнулить BX
CHK_PT:     CMP     DX,0                ;Десятичная точка уже найдена?
            JNZ     RANGE               ; Да. Пропустить дальнейшие проверки
            CMP     BYTE PTR DS:[BP],'.'  ;Десятичная точка?
            JNE     RANGE
            DEC     CX                  ; Да. Уменьшить счетчик и
            MOV     DX,CX               ;  запомнить его в DX
            JZ      END_CONV            ;  Выйти, если CX = 0
            INC     BP                  ;  Продвинуть указатель
RANGE:      CMP     BYTE PTR DS:[BP],'0'  ;Если символ - не
            JB      NON_DIG               ; цифра ...
            CMP     BYTE PTR DS:[BP],'9'
            JBE     DIGIT
NON_DIG:    MOV     DI,BP               ;  то поместить ее адрес в DI,
            STC                         ;  установить флаг переноса и
            JC      END_CONV            ;  выйти из процедуры
DIGIT:      MOV     SI,10               ;Символ - цифра,
            PUSH    DX
            MUL     SI                  ; поэтому умножить AX на 10,
            POP     DX
            MOV     BL,DS:[BP]          ; извлечь ASCII-код,
            AND     BX,0FH              ; оставить только его младшие биты
            ADD     AX,BX               ; и дополнить частичный результат
            JC      END_CONV            ; Выйти, если он слишком велик
            INC     BP                  ; Если нет, продвинуть указатель
            LOOP    CHK_PT              ; и продолжить
            CLC                         ;Если цикл завершен, обнулить флаг
;                                         переноса
END_CONV:   POP     SI                  ;Восстановить значения регистров
            POP     BX
            POP     BP
            RET                         ; и вернуться к вызвавшей процедуре
CONV_AB     ENDP
```

Значение регистра DX определяет характеристику результата. Оно показывает, каким *масштабным коэффициентом* надо воспользоваться, если требуется оперировать преобразованными числами с различным числом знаков после десятичной точки. Содержимое регистра DX может изменяться от 0 (если результатом является целое число) до 5 (при чисто дробном результате). Например, если регистр AX содержит 1000H (десятичное значение 4096), а регистр DX – 2, то полученный Вами результат представляет собой десятичное значение 40,96.

Если Вы хотите добавить это значение к ранее полученному результату, при вычислении которого в регистре DX было возвращено 3, то этот результат надо сначала разделить на 10. Аналогично для сложения 40,96 с ранее полученным результатом, при вычислении которого в регистре DX был возвращен 0, надо сначала разделить новый результат на 100.

Флаг переноса CF показывает, произошла ли ошибка при выполнении преобразования. Флаг CF равен 0, если ошибки не было, и равен 1, если процедура ASCII_BIN обнаружила одну из следующих ошибок:

Если строка содержит более семи символов (CX > 7), то регистры AX и DX содержат 0, а регистр DI – 00FFH.

Если процедура ASCII_BIN обнаружила ошибочный символ, то регистр DI содержит значение смещения его адреса.

Если преобразуемое число вышло за пределы допустимого диапазона (меньше числа – 32768 или больше числа 32767), то регистр AX отличен от нуля, а регистр DI содержит 00FFH.

Для проверки правильности результата вызывайте процедуру ASCII_BIN в следующем контексте:

```
            CALL   ASCII_BIN     ;Вызвать процедуру преобразования
            JNC    VALID         ;Ответ допустим?
            OR     DI,DI         ; Нет. Определить вид ошибки
            JNZ    INV_CHAR
            OR     AX,AX
            JNZ    RANGE_ER
            ...                  ;  Строка слишком длинная
            ...
RANGE_ER:   ...                  ;  Число вне допустимого диапазона
            ...
INV_CHAR:   ...                  ;  Недопустимый символ
            ...
VALID:      ...                  ;Ответ допустим
            ...
```

ПРЕОБРАЗОВАНИЕ ДВОИЧНОГО ЧИСЛА В СТРОКУ ASCII -КОДОВ

Перед выдачей на экран или принтер результат надо преобразовать в строку ASCII-кодов. Для преобразования 16-битового двоичного числа в строку из ASCII-кодов требуется программа, которая подсчитывает, сколько единиц, десятков, сотен, тысяч и десятков тысяч содержит преобразуемое число, и преобразует каждый из этих счетчиков в ASCII-код. Полученные ASCII-коды символов можно выводить по мере вычисления или сохранять в памяти в виде строки, чтобы вывести их позже другой программой.

В примере 6.8 показана процедура BIN_ASCII, которая преобразует 16-битовое значение регистра AX в строку ASCII-кодов, помещаемую в память. Чтобы избежать применения нескольких счетчиков, процедура BIN_ASCII последовательно делит содержимое регистра AX на 10 и каждый раз использует полученный остаток для образования строки. Процедура BIN_ASCII возвращает адрес преобразованной строки в регистре BX, а счетчик ее символов – в регистре CX.

ПРИМЕР 6.8. ПРЕОБРАЗОВАНИЕ ДВОИЧНОГО ЧИСЛА В СТРОКУ ASCII-КОДОВ

```
; Эта процедура преобразует двоичное число со знаком в 6-байтовую
; ASCII-строку (знак плюс 5 цифр), находящуюся в сегменте данных
; Преобразуемое число берется из регистра AX, а начальный адрес бу-
; фера в памяти - из регистра BX
; По возвращению регистр BX содержит адрес преобразованной выходной
; строки, а регистр CX - длину строки
; Значения других регистров сохраняются
;
```

```
BIN_ASCII    PROC
             PUSH   DX          ;Сохранить используемые регистры
             PUSH   SI
             PUSH   AX          ;Сохранить двоичное значение
             MOV    CX,6        ;Заполнить буфер пробелами
FILL_BUFF:   MOV    BYTE PTR [BX],
             INC    BX
             LOOP   FILL_BUFF
             MOV    SI,10       ;Приготовиться к делению на 10
             OR     AX,AX       ;Если значение отрицательное,
             JNS    CLR_DVD
             NEG    AX          ; то изменить его знак
CLR_DVD:     SUB    DX,DX       ;Обнулить старшую половину делимого
             DIV    SI          ;Разделить AX на 10
             ADD    DX,'0'      ;Преобразовать остаток в ASCII-цифру
             DEC    BX          ;Попятиться в буфере
             MOV    [BX],DL     ;Занести этот символ в строку
             INC    CX          ;Подсчитывать преобразованные символы
             OR     AX,AX       ;Все сделано?
             JNZ    CLR_DVD     ; Нет. Получить следующую цифру
             POP    AX          ; Да. Взять исходное значение
             OR     AX,AX       ;Оно было отрицательным?
             JNS    NO_MORE
             DEC    BX          ; Да. Занести в строку знак
             MOV    BYTE PTR [BX],'-'
             INC    CX          ;  и увеличить счетчик символов
NO_MORE:     POP    SI          ;Восстановить значения регистров
             POP    DX
             RET                ; и выйти из процедуры
BIN_ASCII    ENDP
```

УПРАЖНЕНИЯ

1. Разработайте процедуру вычисления промежутка времени между двумя нажатиями на клавиши. Обнулите счетчик времени после первого нажатия; прошедшее время будет равно текущему значению счетчика времени после второго нажатия клавиши.

Для определения *минимального* времени между двумя ударами по клавишам вызовите процедуру из предыдущего упражнения и после появления на экране курсора быстро нажмите два раза на клавишу пробела.

3. Наша программа должна изобразить сообщения "Программа сортировки" и "Нажмите любую клавишу для продолжения" на отдельных строках экрана, но вместо этого получилось

```
Программа сортировкиНажмите любую клавишу для продолжения.
```

Почему так произошло?

4. Напишите программу для выдачи сообщения вида

```
Попробуйте снова. У Вас осталось n истребителей.
```

где n — число от 1 до 6, передаваемое через регистр CX.

ГЛАВА 7. ПРОСТЫЕ СПОСОБЫ ПОЛУЧЕНИЯ ГРАФИЧЕСКИХ ИЗОБРАЖЕНИЙ

7.1. РЕЖИМЫ ИЗОБРАЖЕНИЯ

Если Вы приобрели стандартное изделие фирмы IBM, то Ваш дисплей подсоединен к IBM PC через одну из двух плат адаптера. Адаптер монохроматического дисплея и параллельного принтера позволяет подключить к ЭВМ монохроматический дисплей фирмы IBM и принтер с параллельным обменом данными. Адаптер цветного графического монитора позволяет подключить к ЭВМ монитор или стандартный телевизор.

МОНОХРОМАТИЧЕСКИЙ АДАПТЕР

Адаптер монохроматического дисплея и параллельного принтера позволяет изображать только черные и белые символы *текста* (буквы, цифры и знаки), а также псевдографические символы. Адаптер изображает эти символы путем преобразования каждого из 256 ASCII-кодов символов в его образ. Адаптер заполняет на экране решетку из 25 строк и 80 столбцов символами из находящегося на его плате буфера размером в 4 Кбайта.

ЦВЕТНОЙ ГРАФИЧЕСКИЙ АДАПТЕР

Адаптер цветного графического монитора может работать в двух режимах: текстовом (рассмотренном выше) и графическом. Он позволяет получать черно-белые и цветные изображения, используя 16 возможных цветов.

В текстовом режиме адаптер может генерировать изображение с высоким разрешением, позволяющим разместить на экране 25 строк и 80 столбцов, а также изображение с низким разрешением, при котором экран разбивается на 25 строк и 40 столбцов. Плата адаптера содержит 16 Кбайт памяти, в которой помещается до четырех страниц размером 80 * 25 или до восьми страниц размером 40 * 25.

В графическом режиме адаптер разбивает экран на решетку точечных элементов, или *пэлов*. Он обеспечивает разрешающую способность 320 * 200 и 640 * 200 пэлов. При разрешении 320 * 200 каждый пэл может иметь один из четырех цветов на фоне любого из 16 возможных цветов. При разрешении 640 * 200 возможно только черно-белое изображение, поскольку в этом случае все 16 Кбайт памяти, расположенной на плате адаптера, используются для задания одного из двух состояний пэлов (светлый/темный).

Из-за большого числа пэлов и 16 цветов палитры программирование изображения в графическом режиме представляет собой довольно сложную задачу. (Дополнительные сведения о создании изображений в графическом режиме можно найти в техническом руководстве фирмы IBM.) Поэтому мы ограничимся обсуждением программирования изображений в текстовом режиме с помощью псевдографических символов, входящих в набор символов IBM PC. Все процедуры, приведенные в этой главе, рассчитаны на черно-белое изображение на экране с решеткой 80 * 25. Однако Вы сможете без труда модифицировать их для цветных изображений, а также для решетки 40 * 25.

7.2. ИЗОБРАЖАЕМЫЕ СИМВОЛЫ

НАБОР СИМВОЛОВ

Как уже упоминалось в разд. 7.1, адаптер дисплея при работе в текстовом режиме (в котором он находится до тех пор, пока не будет программой переведен в графический режим) формирует символ с помощью преобразования ASCII-кода, находящегося в его буфере, в одно из 256 изображений. В табл. 7.1 показана связь между ASCII-кодами и изображениями символов.

Обратите внимание, что наряду со стандартными буквами английского алфавита, цифрами и знаками набор символов содержит буквы ряда европейских языков, буквы греческого алфавита и математические символы, а также стрелки и множество других символов. Для графических приложений наиболее интересны символы из столбцов 0, 1, A, B и C. Надо признать, что эти простые псевдографические символы не позволяют создавать сложные графические изображения, кото-

Таблица 7.1

Старшая часть

Младшая часть

Десятичное значение		0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
	Шестнадцатеричное значение	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	Пустой символ (0)	►	Пробел	0	@	P	‘	p	Ç	É	á	1/4	└	╨	∝	≡
1	1	☺	◄	!	1	A	Q	a	q	ü	Æ	í	1/2	┴	╤	β	±
2	2	☻	↕	"	2	B	R	b	r	é	FE	ó	3/4	┬	╥	γ	≥
3	3	♥	‼	#	3	C	S	c	s	â	ô	ú	│	├	╙	π	≤
4	4	♦	¶	$	4	D	T	d	t	ä	ö	ñ	┤	─	╘	Σ	⌠
5	5	♣	§	%	5	E	U	e	u	à	ò	Ñ	╡	┼	╒	σ	⌡
6	6	♠	▬	&	6	F	V	f	v	å	û	ª	╢	╞	╓	μ	÷
7	7	Гудок •	↨	’	7	G	W	g	w	ç	ù	º	╖	╟	╫	τ	≈
8	8	◘	↑	(	8	H	X	h	x	ê	ÿ	¿	╕	╚	╪	Φ	°
9	9	○	↓	)	9	I	Y	i	y	ë	Ö	⌐	╣	╔	┘	θ	•
10	A	◙	→	*	:	J	Z	j	z	è	Ü	¬	║	╩	┌	Ω	·
11	B	♂	esc ←	+	;	K	[	k	{	ï	¢	½	╗	╦	█	δ	√
12	C	♀	∟	,	<	L	\	l	¦	î	£	¼	╝	╠	▄	∞	η
13	D	♪	↔	—	=	M	]	m	}	ì	¥	¡	╜	═	▌	∅	2
14	E	♫	▲	.	>	N	∧	n	∼	Ä	Pts	«	╛	╬	▐	∈	■
15	F	☼	▼	/	?	O	_	o	△	Å	ƒ	»	┐	╧	▀	∩	Пустой символ (FF)

рые позволили бы Вам конкурировать с Диснеем или Жоржем Люка, но они достаточно приемлемы для программ, демонстрирующих динамику сбыта продукции, игровых и рекламно-коммерческих программ.

АТРИБУТЫ

Кроме восьмибитового кода символа каждой позиции экрана приписан также восьмибитовый атрибут. Атрибут задает цвета символа и его фона, интенсивность цветов, а также характер свечения символа – постоянный или мерцающий.

При включении питания ЭВМ плата адаптера устанавливает для всего экрана нормальную интенсивность свечения и немерцающее изображение белых символов на черном фоне. Однако задавая новые значения атрибутов, можно изменять эти первоначально установленные режимы изображения в любой требуемой позиции экрана.

В табл. 7.2 показаны форматы байта атрибута при черно-белом изображении символов. В гл. 2 технического руководства фирмы IBM описаны форматы атрибутов при цветном изображении символов.

Таблица 7.2

Байт атрибута для черно-белого изображения										
Режим изображения	Позиция бита								Цвет символа	Цвет фона
	7	6	5	4	3	2	1	0		
Нормальный	B	0	0	0	I	1	1	1	Белый	Черный
Инвертированный	B	1	1	1	I	0	0	0	Черный	Белый
Изображение отсутствует (черный фон)	B	0	0	0	I	0	0	0	Белый	Черный
Изображенив отсутствует (белый фон)	B	1	1	1	I	1	1	1	Белый	Черный

I = 0 Нормальная интенсивность свечения
= 1 Высокая интенсивность свечения

B = 0 Немерцающее изображение
= 1 Мерцающве изображение

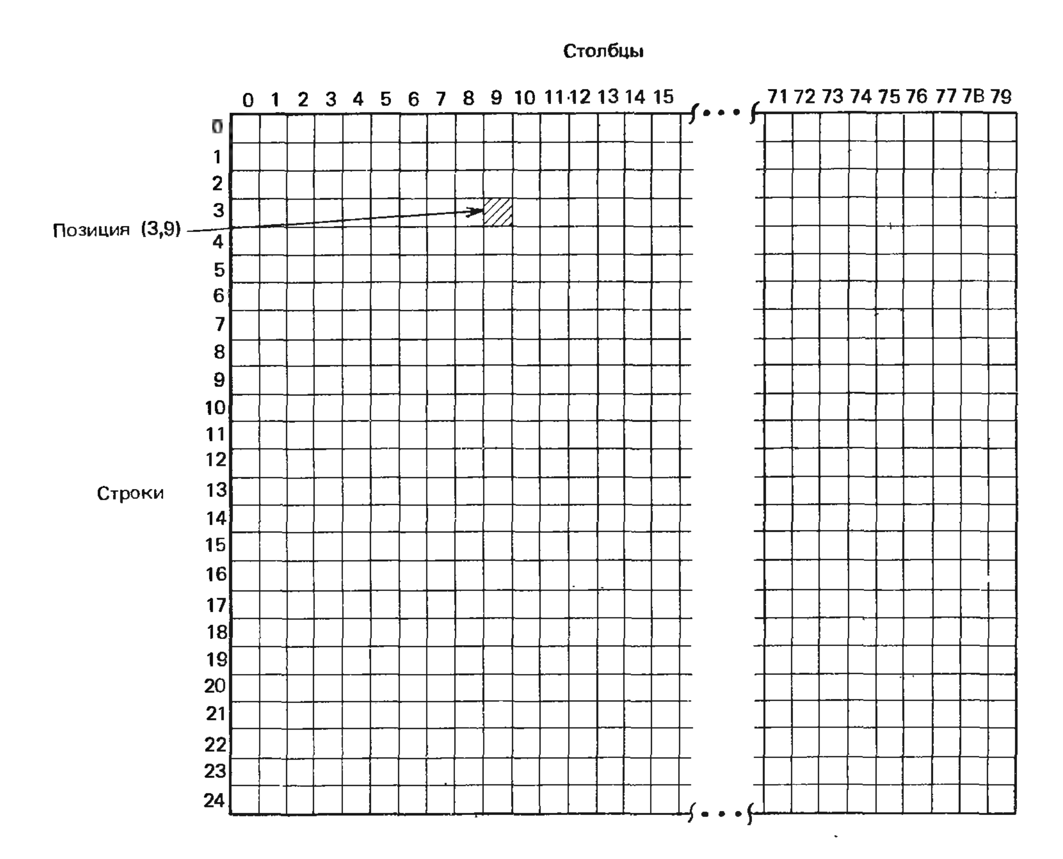


Рис. 7.1. Текстовая решетка экрана 80*25

КОМАНДЫ ДИСПЛЕЯ

Как упоминалось в разд. 7.1, все приведенные в этой главе процедуры изображают черные или белые символы в решетке из 80 столбцов и 25 строк. В решетке 80*25 каждая позиция имеет вертикальную координату (номер строки или линии) от 0 до 24 и горизонтальную координату (номер столбца) от 0 до 79. Следовательно, для изображения символа в десятичной позиции четвертой строки надо задать координату строки со значением 3 и координату столбца со значением 9. Для удобства мы будем пользоваться обозначением (3,9). На рис. 7.1 показано положение этой позиции в решетке.

Каким образом можно обеспечить взаимодействие с экраном? Мы уже встречались с процедурами, выполнявшими такие операции. К ним относятся процедуры, вызываемые с помощью прерывания типа 10 (видеооперации ввода-вывода), которые были описаны в разд. 6.2 и приведены в табл. 6.2. Для удобства в табл. 7.3 показано подмножество табл. 6.2, включающее только те операции, которые относятся к черно-белому текстовому режиму и решетке 80 * 25. Если у Вас монохроматический дисплей, имеющий только одну страницу и один режим изображения (текстовый, черно-белый, с решеткой 80 * 25), то не обращайте внимания на команды и пояснения, набранные курсивом.

Кстати, заметьте, что в табл. 7.3 описаны две отдельные процедуры вывода. Процедура, вызываемая при (AH) = 9 выводит символ и атрибут в текущую позицию курсора, а процедура, вызываемая при (AH) = 10 выводит один только символ и оставляет неизменным атрибут, который был задан ранее для этой позиции решетки. Вторая процедура особенно удобна при выводе черно-белого изображения, когда атрибут меняется редко.

ПРОСТЫЕ ПРИЕМЫ ПОСТРОЕНИЯ ИЗОБРАЖЕНИЙ

Теперь мы может приступить к генерации изображений в текстовом режиме. Для начала изобразим на экране диагональную *линию* с низким разрешением. Эта линия должна начинаться в левом верхнем углу экрана и пересекать его по диагонали, "продвигаясь" на каждом шаге на одну строку и один столбец.

Для образования линии используем "улыбающуюся рожицу" (код 02 в табл. 7.1). Таким образом, первая рожица должна появиться в позиции экрана с координатами (0,0), вторая – в позиции (1,1) и т. д.

В примере 7.1 показана процедура DIAG_LINE, которая вычерчивает эту диагональную линию. Она использует четыре процедуры из числа тех, что могут быть вызваны с помощью прерывания типа 10. Процедура, вызываемая при (AH) = 15, считывает номер страницы дисплея в регистр BH; при (AH) = 0 устанавливается черно-белый текстовый режим с решеткой 80 * 25; при (AH) = 2 перемещается курсор, а при (AH) = 10 на экране изображается символ.

Таблица 7.3. **Операции, инициируемые прерыванием типа 10 в черно-белом текстовом режиме с решеткой 80 * 25**

Регистр AH	Операция	Дополнительные входные регистры	Выходные[1] регистры
	Процедуры интерфейса дисплея		
0	Задание режима изображения	(AL) = 2 80 * 25, черно-белый текстовый режим	Не используются
2	Перемещение курсора в заданную позицию	(DH, DL) = (строка, столбец) (0 – 24, 0 – 79) (BH) = номер страницы (0 – 4)	Не используются
3	Чтение положения курсора	(BH) = номер страницы (0 – 4)	(DH, DL) = строка, столбец курсора (CH, CL) = текущий режим курсора
5	Задание новой активной страницы дисплея	(AL) = новый номер страницы (0 – 3)	Не используются
6	Прокрутка активной страницы вверх	(AL) = число строк Строки, появляющиеся внизу окна, заполняются пробелами. При (AL) = 0 пробелами заполняется все окно. (CH, CL) = строка, столбец верхнего левого угла прокручиваемого окна (DH, DL) = строка, столбец нижнего правого угла прокручиваемого окна (BH) = атрибут, используемый при изображении строки пробелов	Не используются
7	Прокрутка активной страницы вниз	Те же, что и выше, но строки пробелов входят в окно сверху	Не используются

Процедуры обработки символов			
8	Чтение символа, находящегося в текущей позиции курсора и его атрибута	(BH) = номер страницы дисплея (0 – 3)	(AL) = считанный символ (AH) = атрибут этого символа
9	Запись символа и нового атрибута в текущую позицию курсора	(BH) = номер страницы дисплея (0 – 3) (BL) = атрибут символа (CX) = счетчик записываемых символов (AL) = записываемый символ	Не используются
10	Запись символа без изменения атрибута в текущую позицию курсора	(BH) = номер страницы дисплея (0 – 3) (CX) — счетчик записываемых символов (AL) = записываемый символ	Не используются
Процедура *ASCII-телетайп* для вывода			
14	Вывод символа на экран и перемещение курсора в следующую позицию	(AL) = записываемый символ (BH) = номер страницы дисплея (0 – 3)	Не используются
Чтение видеостатуса			
15	Чтение текущего видеостатуса	Не используются	(AL) = текущий режим (см. (AH) = 0 для разъяснения) (AH) = число столбцов на экране (BH) = текущая страница дисплея

[1] Наряду с возвращением значений в перечисленных здесь регистрах эти процедуры сохраняют значения регистров CS, SS, DS, ES, BX, CX и DX. Значения всех остальных регистров следует считать уничтоженными.

```
; Эта процедура изображает диагональную линию из "улыбающихся
; рожиц", начинающуюся в левом верхнем углу экрана
; Значения всех регистров сохраняются
;
DIAG_LINE   PROC
            PUSH   AX            ;Сохранить значения регистров
            PUSH   BX
            PUSH   CX
            PUSH   DX
            MOV    AH,15         ;Загрузить в BH номер активной
;                                  страницы дисплея
            INT    10H
            MOV    AH,2          ;Выбрать текстовый режим 80x25, ч/б
            INT    10H
            MOV    CX,1          ;Счетчик символов = 1
            MOV    DX,0          ;Начать со строки 0 в столбце 0
SET_CRSR:   MOV    AH,2          ;Переместить курсор в следующую позицию
            INT    10H
            MOV    AL,2          ;Изображаемый символ - улыбающаяся рожица
            MOV    AH,10         ;Вывести символ на экран
            INT    10H
            INC    DH            ;Указать на позицию в следующей строке
            INC    DL            ; и в следующем столбце
            CMP    DH,25         ;Дошли до низа экрана?
            JNE    SET_CRSR
            POP    DX            ; Да. Восстановить значения регистров
            POP    BX
            POP    AX
            RET                  ;  и выйти из процедуры
DIAG_LINE   ENDP
```

7.3. ОСНОВЫ ОЖИВЛЕНИЯ ИЗОБРАЖЕНИЙ

Процедура из примера 7.1 выдает статичное изображение, детали которого неподвижны. "Рожицы" помещаются в заданных местах и улыбаются, как бы говоря Вам : "Прекрасный денек!" Однако во многих приложениях требуется, чтобы образы двигались по экрану. Иначе говоря, их надо *оживлять*.

Получить живые картинки не так трудно, как это может показаться. Действительно, для этого требуется выполнить всего пять следующих шагов:

1. Начертить образ(ы) на экране.
2. Выдержать паузу, чтобы успеть увидеть образ.
3. Стереть образ.
4. Изменить координаты строки и столбца начала образа.
5. Повторить этот процесс с шага 1.

Для генерации паузы (шаг 2) можно использовать процедуру DELAY, разработанную в примере 6.2.

Для стирания образа с экрана надо либо *заполнить* занимаемую им часть экрана пустыми символами, либо *перечертить* его черным цветом. Для этого надо либо выдать на экран пустые символы с помощью процедуры, вызываемой при (AH) = 9, либо присвоить символам образа атрибут невидимости ((BL) = 0) с помощью процедуры, вызываемой при (AH) = 10.

В примере 7.2 показана другая процедура вычерчивания линии, которая передвигает "улыбающуюся рожицу" вниз по диагонали с интервалами в 1/2 с. Основная процедура MOVE_FACE всего лишь на четыре команды длиннее, чем процедура DIAG_LINE из примера 7.1. Она вызывает процедуру DLY_HALF, обеспечивающую паузу в 1/2 с, и стирает "рожицу", посылая на экран код пустого символа (0).

Вы можете поэкспериментировать с меньшими или большими паузами. А если Вы хотите увидеть (а точнее, *не увидеть*) движение "рожицы" с максимальной скоростью, то паузу можно исключить вовсе.

ПРИМЕР 7.2. ОЖИВЛЕНИЕ ИЗОБРАЖЕНИЯ ДИАГОНАЛЬНОЙ ЛИНИИ

```
; Эта процедура перемещает "улыбающуюся рожицу" по диагонали сверху
; вниз, начиная с левого верхнего угла. Интервал между перемещениями
; составляет 1/2 с. Значения всех регистров сохраняются
;
            EXTRN  DELAY:FAR
MOVE_FACE   PROC
            PUSH   AX          ;Сохранить значения регистров
            PUSH   BX
            PUSH   CX
            PUSH   DX
            MOV    AH,15       ;Загрузить в BH номер активной
                                 страницы дисплея
            INT    10H
            MOV    AH,2        ;Выбрать текстовый режим 80x25, ч/б
            INT    10H
            MOV    CX,1        ;Счетчик символов = 1
            MOV    DX,0        ;Начать со строки 0 в столбце 0
SET_CRSR:   MOV    AH,2        ;Переместить курсор в следующую позицию
            INT    10H
            MOV    AL,2        ;Изображаемый символ - улыбающаяся рожица
            MOV    AH,10       ;Вывести символ на экран
            INT    10H
            CALL   DLY_HALF    ;Выждать полсекунды
            SUB    AL,AL       ; и стереть рожицу
            MOV    AH,10
            INT    10H
            INC    DH          ;Указать на позицию в следующей строке
            INC    DL          ; и в следующем столбце
            CMP    DH,25       ;Дошли до низа экрана?
            JNE    SET_CRSR
            POP    DX          ; Да. Восстановить значения регистров
            POP    BX
            POP    AX
            RET                ;  и выйти из процедуры
DIAG_LINE   ENDP
;
; Следующая процедура генерирует паузу в 1/2 с
;
DLY_HALF    PROC
            SUB    AL,AL       ;Положить минуты = 0.
            SUB    BH,BH       ;Положить секунды = 0
            MOV    BL,50       ;Положить сотые = 50 (1/2 с)
            CALL   DELAY
            RET
DLY_HALF    ENDP
```

Манипулировать образами, составленными из нескольких символов, почти столь же легко, как и теми, что представляют собой один символ, например "улыбающуюся рожицу". Но построение изображения из нескольких символов требует выполнения нескольких операций вывода данных – по одной для каждого символа.

Если образ состоит всего из двух или трех символов и ничего другого Вам не требуется изобразить, то проще всего запрограммировать его изображение с помощью нескольких операций вывода данных. Но если либо образ составлен из многих символов, либо требуется изобразить два или более образа одновременно, то полезно поместить параметры изображения в *таблицу образа.*

Таблица образа содержит коды символов, атрибуты и смещения номеров строки и столбца каждого символа, входящего в состав образа. Вы уже имеете представление о коде символа и его атрибуте. А смещения номеров строки и столбца задают соответственно вертикальную и горизонтальную дистанции между предыдущим и текущим символами. Таким образом, эти числа "говорят" ЭВМ, как строить образ.

В качестве иллюстрации на рис. 7.2 показана "тележка", составленная из семи символов. Корпус составлен (слева направо) из буквы R (код 52H), заштрихованного квадрата (код B1) и двух сплошных квадратов (код DB); изображение первых двух символов инвертировано, а последние два символа выведены в обычном режиме. Колеса (буквы O, код 4F) и "улыбающаяся рожица" (код 02) также выведены в нормальном режиме. Этот образ сконструировал сын автора – Райан.

В скобках на рис. 7.2 указаны относительные смещения номеров строки и столбца. Первой изображается буква R, поэтому смещения равны (0,0). Затем изображается заштрихованный квадрат. Он находится в той же строке, что и буква R, но смещен на один столбец вправо. Поэтому его смещения равны (0,1). Символы передней части корпуса также имеют смещения (0,1).

Затем изображается переднее колесо, имеющее смещения (1,0) по отношению к крайнему правому символу корпуса, и заднее колесо, имеющее смещения (0, – 3) по отношению к переднему колесу. Наконец, "улыбающаяся рожица" имеет смещения (– 2,1) по отношению к заднему колесу.

Каждый параметр образа занимает один байт, поэтому таблица образа должна содержать по четыре байта на каждый символ. Если Вам требуется оперировать несколькими таблицами образов, то каждая таблица должна включать байт, содержащий число символов. На рис. 7.3 показана таблица образа для только что описанной "тележки".

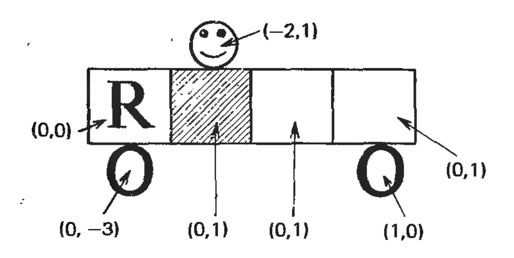


Рис. 7.2. Смещение строк и столбцов "тележки", составленной из символов

```
CAR     DB   7                    ;Образ содержит 7 символов
        DB   52H,70H,0,0          ;Задний символ корпуса
        DB   0B1H,70H,0,1         ;Место водителя
        DB   0DBH,7,0,1           ;Следующий символ корпуса
        DB   0DBH,7,0,1           ;Передний символ корпуса
        DB   4FH,7,1,0            ;Переднее колесо
        DB   4FH,7,0,-3           ;Заднее колесо
        DB   2,7,-2,1             ;Улыбающаяся рожица
```

Рис. 7.3. Таблица образа "тележки", составленной из символов

УНИВЕРСАЛЬНАЯ ПРОЦЕДУРА ИЗОБРАЖЕНИЯ

Ваша программа вывода изображения может состоять из процедуры, которая загружает параметры таблицы образа в соответствующие регистры, а затем дает команду вывода данных. Эта процедура помещается в цикл и должна повторяться до тех пор, пока все символы таблицы образа не будут выведены.

В примере 7.3 показана процедура, которая стирает весь экран с помощью прокрутки активной страницы вверх, инициируемой с помощью прерывания типа 10.

После очистки экрана можно вывести образ на экран, вызывая процедуру из примера 7.4. Например, следующая последовательность команд выводит на экран нашу "тележку", располагая ее корпус в строке 20:

```
MOV     CH,0              ;Стереть экран
MOV     DH,24
CALL    CLEAR_SCREEN
LEA     DI,CAR            ;DI указывает на таблицу с образом тележки
MOV     DH,20             ;Корпус начинается на строке 20 ,
MOV     DL,0              ; в столбце 0
CALL    DSPLY_SHAPE       ;Начертить тележку
```

ПРИМЕР 7.3. ОЧИСТКА ЭКРАНА

```
; Эта процедура стирает экран, давая каждой строке атрибут
; нормального белого изображения на черном фоне
; Значения всех регистров сохраняются
;
CKEAR_SCREEN   PROC
               PUSH   AX        ;Сохранить значения регистров
               PUSH   BX
               PUSH   CX
               POP    DX
               MOV    AH,6      ;Стереть экран в режиме
               MOV    AL,0      ; прокрутки сверху
               MOV    CX,0      ;Стереть со строки 0 и столбца 0
               MOV    DH,24     ; по строку 24
               MOV    DL,79     ; и столбец 79
               MOV    BH,7
               INT    10H
               POP    DX        ;Восстановить значения регистров
               POP    CX
               POP    BX
               POP    AX
               RET              ; и выйти из процедуры
CLEAR_SCREEN   ENDP
```

```
; Эта процедура воспроизводит на экране образ по таблице, находя-
; щейся в сегменте данных
; Перед вызовом в регистр DI заносится адрес таблицы образа, а  в
; регистры DH и DL - координаты строки и столбца первого символа
; образа
; Таблица образа содержит байт счетчика символов, за которым ука-
; зываются код, атрибут, сдвиг по строке и сдвиг по столбцу для
; каждого символа образа
; Значения регистров не изменяются
;
DSPLY_SHAPE    PROC
               PUSH   AX            ;Сохранить значения регистров
               PUSH   BX
               PUSH   CX
               PUSH   DX
               PUSH   DI
               MOV    AH,15         ;Загрузить в BH номер активной
;                                     страницы дисплея
               INT    10H
               SUB    CH,CH         ;Обнулить старший байт счетчика
               MOV    CL,[DI]       ;CL содержит счетчик символов
               INC    DI            ;DI указывает на первый символ
NEXT_CHAR:     ADD    DH,[DI+2]     ;Модифицировать указатели строки
               ADD    DL,[DI+3]     ; и столбца
               MOV    AH,2          ;Переместить курсор
               INT    10H
               MOV    AL,[DI]       ;Извлечь код символа
               MOV    BL,[DI+1]     ; и его атрибут
               PUSH   CX            ;Сохранить счетчик символов
               MOV    CX,1          ;Изобразить символ на экране
               MOV    AH,9
               INT    10H
               POP    CX            ;Восстановить счетчик символов
               ADD    DI,4          ;DI указывает на блок данных
;                                     следующего символа
               LOOP   NEXT_CHAR     ;Когда все символы будут изображены,
               POP    DI            ; восстановить значения регистров
               POP    DX
               POP    BX
               POP    AX
               RET                  ;  и выйти из процедуры
DSPLY_SHAPE    ENDP
```

ПРОГРАММА ДВИЖЕНИЯ ОБРАЗА

Можно оживить и сложный образ, например нашу "тележку", методом, который применялся при оживлении образа, состоящего из одного символа. Чтобы передвигать по экрану образ, составленный из нескольких символов, надо вычертить его, выждать некоторое время, стереть образ и перечертить его в новом месте.

Как уже упоминалось, для стирания образа можно либо очистить ту часть экрана, в которой он находится, либо перечертить его черным цветом. Но в случае наличия на экране нескольких образов позаботьтесь о том, чтобы иметь информацию о текущем местонахождении стираемого образа. При движении образа это может представлять некоторую трудность.

В примере 7.5 показана процедура, обеспечивающая движение образа через экран по горизонтали. Основная процедура MOVE_SHAPE чертит образ на экране с помощью того же общего подхода, что и процедура DSPLY_SHAPE. Пауза (в дан-

ном случае длительностью 1/4 с) и очистка экрана осуществляются двумя дополнительными процедурами: DLY_QRTR и ERASE.

Процедура ERASE стирает образ, перечерчивая его с нулевым значением всех атрибутов. Тем самым обеспечивается черное изображение на черном фоне (т. е. *отсутствие изображения*). Кроме того, процедура ERASE по окончании операции стирания увеличивает номер столбца – переменную COL_NO из сегмента данных. Так как правый крайний столбец экрана имеет номер 79, то процедура MOVE_SHAPE заканчивает работу, как только какой-либо из символов, составляющих образ, получит номер столбца, больший или равный 80.

ПРИМЕР 7.5. ОЖИВЛЕНИЕ ОБРАЗА, СОСТОЯЩЕГО ИЗ НЕСКОЛЬКИХ СИМВОЛОВ

```
; Эта процедура перемещает образ горизонтально по экрану, делая
; между перемещениями паузу в 1/4 с
; Перед вызовом в регистре DI должен находиться адрес таблицы образа,
; а в регистрах DH и DL - координаты строки и столбца первого символа
; образа
; Значения регистров не изменяются
;
; Указанные ниже рабочие переменные должны быть определены в сегменте
; данных
;
CHAR_CNT   DW ?
POINTER    DW ?
LINE_NO    DW ?
COL_NO     DW ?
;
; Ниже приводится основная процедура
;
               EXTRN  DELAY:FAR
MOVE_SHAPE     PROC
               PUSH   AX             ;Сохранить значения регистров
               PUSH   BX
               PUSH   CX
               PUSH   DX
               PUSH   DI
               MOV    AH,15          ;Загрузить в BH номер активной
;                                      страницы
               INT    10H
               SUB    CH,CH          ;Обнулить старший байт счетчика
               MOV    CL,[DI]        ;
               INC    DI             ;DI указывает на первый символ
               MOV    CHAR_CNT,CX    ;Сохранить счетчик символов,
               MOV    POINTER,DI     ; указатель таблицы образа
               MOV    LINE_NO,DL     ; и координаты
               MOV    COL_NO,DL
PLOT_NEXT:     ADD    DH,[DI+2]      ;Модифицировать указатели строки
               ADD    DL,[DI+3]      ; и столбца
               CMP    DL,80          ;Символ вышел за пределы экрана?
               JB     MOV_CRSR
               CALL   ERASE          ; Да. Стереть образ и выйти
               POP    DI             ;  из процедуры
               POP    DX
               POP    CX
               POP    BX
               POP    AX
               RET
MOV_CRSR:      MOV    AH,2           ;Переместить курсор
               INT    10H
               MOV    AL,[DI]        ;Извлечь код символа
               MOV    BL,[DI+1]      ; и его атрибут
               PUSH   CX             ;Сохранить счетчик символов
               MOV    CX,1           ;Изобразить символ на экране
               MOV    AH,9
```

```
                INT     10H
                POP     CX              ;Восстановить счетчик символов
                ADD     DI,4            ;DI указывает на блок данных
                                          следующего символа
                LOOP    PLOT_NEXT       ;Когда все символы будут изображены,
                CALL    DLY_QRTR        ; выждать 1/4 с
                CALL    ERASE           ; и затем стереть образ
                MOV     CX,CHAR_CNT     ;Загрузить заново счетчик символов,
                MOV     DI,POINTER      ; указатель таблицы образа
                MOV     DH,LINE_NO      ; и номер строки
                INC     COL_NO          ;Указать на следующий столбец
                MOV     DL,COL_NO
                JMP     SHORT PLOT_NEXT
MOVE_SHAPE      ENDP
;
; Эта процедура стирает образ, перечерчивая его с атрибутом = 0
;
ERASE           PROC
                MOV     CX,CHAR_CNT     ;Загрузить заново счетчик символов,
                MOV     DI,POINTER      ; указатель таблицы образа,
                MOV     DH,LINE_NO      ; номер строки
                MOV     DL,COL_NO       ; и номер столбца
ERASE_NEXT:     ADD     DH,[DI+2]
                ADD     DL,[DI+3]
                MOV     AH,2
                INT     10H
                MOV     AL,[DI]         ;Для стирания символа используйте
                MOV     BL,0            ; атрибут = 0
                PUSH    CX
                MOV     CX,1
                MOV     AH,9
                INT     10H
                POP     CX
                ADD     DI,4
                LOOP    ERASE_NEXT
                MOV     CX,CHAR_CNT     ;Загрузить заново счетчик символов,
                MOV     DI,POINTER      ; указатель таблицы образа
                MOV     DH,LINE_NO      ; и номер строки
                INC     COL_NO          ;Указать на следующий столбец
                MOV     DL,COL_NO
                RET                     ; и выйти в вызвавшую процедуру
ERASE           ENDP
;
; Эта процедура генерирует паузу в 1/4 с
;
DLY_QRTR        PROC
                SUB     AL,AL           ;Положить минуты = 0
                SUB     BH,BH           ;Положить секунды = 0
                MOV     BL,25           ;Положить сотые = 25 (1/4 с)
                CALL    DELAY
                RET
DLY_QRTR        ENDP
```

Хотя приведенная выше процедура обеспечивает только горизонтальное движение образа, ее нетрудно модифицировать для движения по вертикали или по диагонали. Для этого после каждой последовательности начертить-выждать-стереть достаточно вместо координаты столбца изменять координату строки или координаты и строки, и столбца.

УПРАЖНЕНИЯ

1. В дополнение к движению образа по экрану можно ввести и другой элемент оживления, заставляя образ *изменяться* по мере его движения. Чтобы установить, как это делается, разработайте процедуру, которая перемещает "птичку" вдоль экрана так, что ее изображение изменяется со строчной буквы "v" (код

76 в табл. 7.1) на "тире" (код 0С4Н) и наоборот при перемещении в следующий столбец. Начните изображать "птичку" в столбце 0 строки 20 и задерживайте ее в каждом столбце на 0,1 с.

2. Игровые программы часто пользуются случайными числами для перемещения образов по экрану. Чтобы познакомиться с этим поближе, разработайте программу, которая изображает "улыбающуюся рожицу" (код символа 2) в столбце 0 случайно выбранной строки, а затем перемещает ее вдоль экрана.

"Рожица" должна продвигаться за прием на один столбец, но при этом может перепрыгивать на одну строку вверх или вниз в зависимости от того, что выдаст генератор случайных чисел: 0 (вниз на одну строку), 1 (вверх на одну строку) или 2 (та же строка).

Операция вывода изображения должна завершиться, если "рожица" пересечет строку 0, строку 24 или столбец 79. Для получения как номера начальной строки, так и индикатора вверх-вниз-вдоль воспользуйтесь прерыванием типа 1А (время дня); см. пример 6.1 в разд. 6.2.

ГЛАВА 8. ДА БУДЕТ ЗВУК!

В языке Бейсик для IBM PC предусмотрен оператор SOUND, который позволяет генерировать звуки с помощью встроенного динамика. Оператор SOUND имеет два операнда: частоту и длительность звучания. Частота может изменяться от 37 до 32767 Гц (колебаний в секунду), а длительность – от 0 до 65535 отсчетов таймера. (Эти отсчеты регистрируются при обработке прерываний от таймера 8253, обсуждавшихся в гл. 6. Напомним, что эти прерывания происходят примерно 18,2 раза в секунду.)

В этой главе мы разработаем аналог оператора SOUND на языке ассемблера и попробуем музицировать с помощью динамика. Однако наша программа на языке ассемблера будет несколько гибче оператора SOUND, позволяя задавать длительности звучания с шагом 1/100 с вместо шага 1/20 с, обеспечиваемого этим оператором языка Бейсик.

8.1. ПРИНЦИП РАБОТЫ ДИНАМИКА

В IBM PC большинство операций обмена данными регулируется микросхемой *программируемого интерфейса периферийных устройств* (ПИПУ) 8255, расположенной на системной плате. Эта микросхема содержит три 8-битовых регистра; два из них используются для операций ввода данных, а один – для вывода. Входным регистрам присвоены номера портов ввода-вывода 60Н и 62Н; выходному регистру присвоен номер порта ввода-вывода 61Н.

Как показано на рис. 8.1, динамиком можно управлять в двух режимах, задаваемых значениями двух битов выходного регистра ПИПУ. Если бит 0 равен 1, то

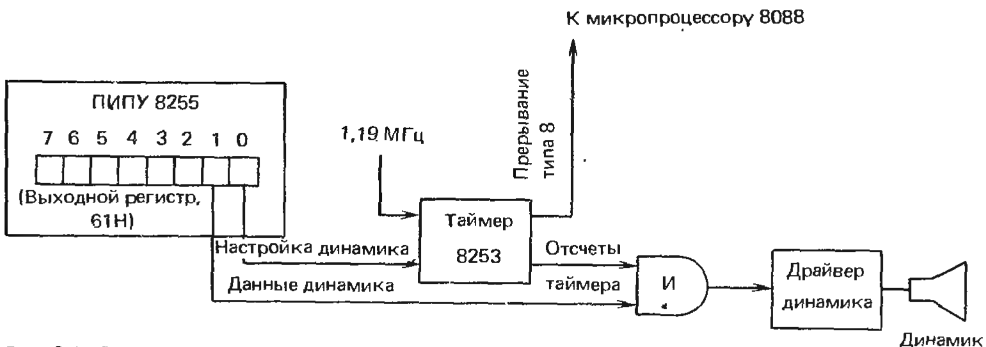


Рис. 8.1. Система управления динамиком

микросхема таймера 8253 задает частоту звучания динамика. Если бит 1 равен 1, то динамик начинает издавать звук и продолжает звучать до тех пор, пока бит 1 не станет равным 0. Программы этой главы ориентированы на использование таймера 8253.

8.2. ПРОГРАММИРОВАНИЕ ДИНАМИКА

Чтобы избежать повторного изобретения колеса, мы положили в основу нашей процедуры управления динамиком процедуру BEEP, которую система BIOS использует для генерации звуков в случае, если тестирование системы в момент включения обнаруживает аппаратную ошибку. Начало процедуры BEEP имеет смещение адреса E666 в системе BIOS IBM PC и FA08 в системе BIOS IBM XT.

ПРОЦЕДУРА BEEP СИСТЕМЫ BIOS

Процедура BEEP генерирует тон с частотой 1000 Гц в течение одного или нескольких интервалов в 1/2 с, число которых задается в регистре BX. Значение этого счетчика интервалов (1 для короткого гудка и 6 для длинного гудка) задается вызывающей процедурой ERROR_BEEP.

Процедура BEEP настраивает таймер 8253 для генерации частотного выхода, а затем посылает значение *делителя* 533H в таймер 2 микросхемы 8253. Тем самым задается частота 1000 Гц. После этого процедура BEEP изменяет значения битов 0 и 1 регистра AL на 1 и переписывает содержимое регистра AL в выходной регистр (порт ввода-вывода 61H) микросхемы 8255, в результате чего инициируется звучание динамика. Последующий цикл сохраняет звучание в течение одного или нескольких интервалов в 1/2 с в зависимости от значения регистра BX.

БОЛЕЕ УНИВЕРСАЛЬНЫЙ ГЕНЕРАТОР ЗВУКОВ

Процедура BEEP служит весьма удобным прототипом для универсальной процедуры генерации звуков, но ее конструкция должна быть изменена в двух аспектах. Во-первых, процедура BEEP обеспечивает только тон с частотой 1000 Гц; мы же должны иметь возможность генерировать тон любой частоты. Во-вторых, процедура BEEP обеспечивает длительность звучания, кратную 1/2 с, а нам требуются длительности звучания, кратные 10 мс.

Процедура BEEP показывает нам, что загрузка значения 533H (десятичного числа 1331) в таймер 2 приводит к генерации тона с частотой 1000 Гц. Следовательно, для генерации тона другой частоты мы должны умножить 1331 на 1000 и поделить результат на заданную частоту. Если эта частота передается нашей процедуре генерации звуков через регистр DI, то требуемое для таймера 2 значение регистра AX можно получить с помощью последовательности команд

```
MOV DX,14H        ;Делитель таймера =
MOV AX,4F38H      ; 1331000/частота
DIV DI
```

Паузу в 10 мс можно генерировать с помощью двух команд вида

```
          MOV  CX,n
SPKR_ON:  LOOP SPKR_ON
```

где n – такое число, при котором этот фрагмент исполняется 10 мс.

В приложении В показывается, что команда MOV исполняется за четыре такта, а команда LOOP за 17 тактов, если имеет место передача управления, и за 5 тактов в противном случае. Так как передача управления происходит до тех пор, пока значение регистра CX не станет равным 1, т. е. n – 1 раз, то справедливо соотношение

$$(17(n-1)+5+4)(210*10^{-9}) = 0.01,$$

из которого следует, что n = 2801. Таким образом, пауза в 10 мс обеспечивается командами

```
            MOV     CX,2801
SPKR_ON:    LOOP    SPKR_ON
```

После внесения этих изменений в процедуру BEEP мы получим универсальную процедуру, которая генерирует звуки любой частоты (заданной значением регистра DI) и любой длительности (заданной значением регистра BX с шагом 0,01 с). В примере 8.1 показана полученная таким образом процедура SOUND, написанная на языке ассемблера.

Как сказано в комментарии к процедуре SOUND, она может генерировать звуки с частотой от 21 до 65 535 Гц. (Нижний предел 21 Гц является наименьшим значением регистра DI, на которое можно поделить 1331000, не вызвав при этом переполнения.) Верхний предел, конечно же, избыточен, поскольку человеческое ухо не воспринимает звуки с частотой свыше 20 000 Гц. Конечно, это ограничение менее жесткое, чем ограничение от 37 до 32 767 Гц у оператора SOUND языка Бейсик.

Далее, процедуре SOUND можно передавать в качестве длительности звучания значения от 0 до 65 535. Но так как перед проверкой на нуль из значения регистра BX вычитается 1, то передача нуля отвечает длительности 65 536. Следовательно, процедура SOUND обеспечивает длительность звучания от 0,01 с (при (BX) = 1) до 655,36 с (при (BX) = 0).

ПРИМЕР 8.1. ПРОЦЕДУРА ГЕНЕРАЦИИ ЗВУКОВ

```
; Эта процедура заставляет динамик ПЭВМ издать тон заданной
; частоты и заданной длительности
; Перед вызовом загрузите частоту в регистр DI (от 21 до
; 65 535 Гц), а длительность (в сотых долях секунды) -
; в регистр BX (от 0 до 65 535)
; Значения всех регистров сохраняются
;
SOUND           PROC
                PUSH    AX              ;Сохранить значения регистров
                PUSH    BX
                PUSH    CX
                PUSH    DX
                PUSH    DI
                MOV     AL,0B6H         ;Записать в регистр режим таймера
                OUT     43H,AL
                MOV     DX,14H          ;Делитель времени =
                MOV     AX,4F38H        ; 1331000/частота
                DIV     DI
                OUT     42H,AL          ;Записать младший байт счетчика таймера 2
                MOV     AL,AH
                OUT     42H,AL          ;Записать старший байт счетчика таймера 2
                IN      AL,61H          ;Считать текущую установку порта B
                MOV     AH,AL           ; и сохранить ее в регистре AH
                OR      AL,3            ;Включить динамик
                OUT     61H,AL
WAIT:           MOV     CX,2801         ;Выждать 10 мс
```

```
SPKR_ON:        LOOP   SPKR_ON
                DEC    BX              ;Счетчик длительности исчерпан?
                JNZ    WAIT            ; Нет. Продолжить звучание
                MOV    AL,AH           ; Да. Восстановить исходную установку
;                                        порта
                OUT    61H,AL
                POP    DI              ;Восстановить значения регистров
                POP    DX
                POP    CX
                POP    BX
                POP    AX
                RET                    ; и выйти из процедуры
SOUND           ENDP
```

8.3. МУЗЫКА, МУЗЫКА, МУЗЫКА

С помощью нашей универсальной процедуры генерации звуков можно заставить динамик издавать мелодии. Однако для этого мы должны знать частоты звучания музыкальных нот.

На рис. 8.2 показана часть клавиатуры пианино, охватывающая две октавы (в октаве восемь нот), и указана частота звучания каждой ноты в герцах. Первая октава начинается с нижней ноты *до* и кончается на средней ноте *до*, а вторая начинается со средней ноты *до* и кончается на верхней ноте *до*.

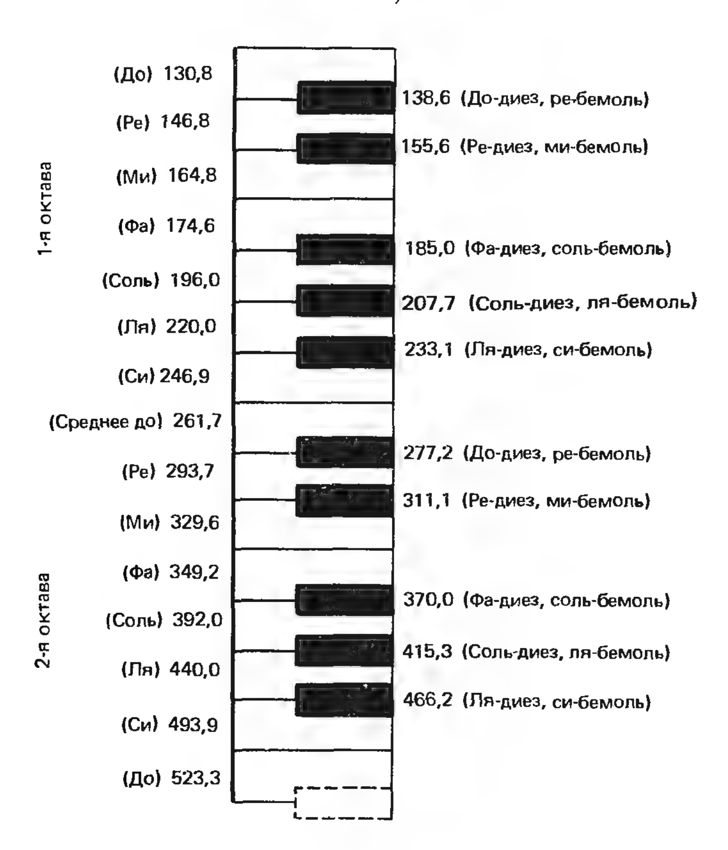
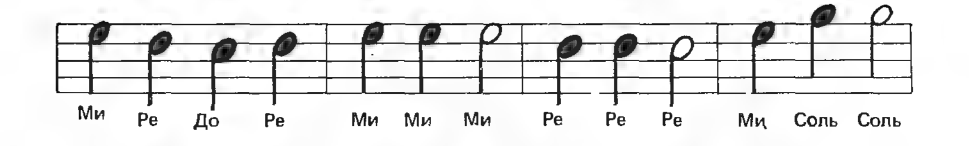
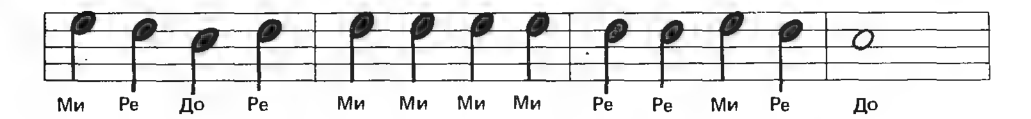


Рис. 8.2. Две октавы на клавиатуре пианино






```
MARY_FREQ  DW  330,  294,  262,  294,  3 DUP (330)  ;  Такты 1 и 2
           DW  294,  294,  294,  330,  392,  392    ;  Такты 3 и 4
           DW  330,  294,  262,  294,  4 DUP (330)  ;  Такты 5 и 6
           DW  294,  294,  330,  294,  262,  0FFFFH ;  Такты 7 и 8

MARY_TIME  DW  6 DUP (25),  50                      ;  Такты 1 и 2
           DW  2 DUP (25,  25,  50)                    Такты 3 и 4
           DW  12 DUP (25),  100                    ;  Такты 5 – 8
```

Рис. 8.3. Мелодия "У Мэри была маленькая овечка"

На этом рисунке мы пометили каждую клавишу стандартным именем соответствующей ей ноты. Белые клавиши (*до* – *си*) соответствуют *основным* нотам, а черные – *производным*, получаемым из основных добавлением слов *бемоль* или *диез*. Черные клавиши издают звуки на полутон выше (*диез*) или полутон ниже (*бемоль*) по сравнению с соседними белыми клавишами.

Теперь мы вооружены всеми необходимыми сведениями для разработки программы генерации мелодий. Что должна делать эта программа? Она должна извлекать из двух таблиц серию частот и длительностей нот и повторять вызовы процедуры SOUND для получения от динамика звучания нот.

Значения частот, конечно, должны соответствовать значениям, показанным на рис. 8.2. Длительности нот зависят от требуемого темпа и числа четвертых отсчетов в ее звучании. *Целая нота* продолжается четыре четвертных отсчета, *половинная нота* – два отсчета, *четвертная* – один отсчет, *восьмая нота* – половину отсчета и т. д.

В качестве простой иллюстрации на рис. 8.3 показана нотная запись мелодии "У Мэри была маленькая овечка" и соответствующие ей таблицы частот и длительностей нот. Большинство нот в ней – четвертные, но второй, третий и четвертый такты заканчиваются половинными нотами (соответственно *ми*, *ре* и *соль*), а последний такт состоит из целой ноты (*до*).

При заполнении таблицы длительностей нот мы произвольно приписали целой ноте длительность в 1 с. Следовательно, половинная нота будет звучать 1/2 с, а четвертная – 1/4 с. Последнее значение в таблице частот (0FFFFH) не имеет отношения к нотам; оно является признаком, по которому программа генерации мелодии прекращает свою работу.

ПРОЦЕДУРА ИСПОЛНЕНИЯ МЕЛОДИИ

Имея процедуру SOUND и таблицы частот и длительностей нот, мы можем приступить к разработке процедуры, обрабатывающей эти таблицы и вызывающей процедуру SOUND, чтобы "исполнить" мелодию. В примере 8.2 показана процедура PLAY, выполняющая эту задачу. В ней предполагается, что таблицы частот и длительностей нот находятся в сегменте данных, а их адреса передаются соответственно в регистрах SI и BP.

МОДУЛЬ МЕЛОДИИ

Нам нужен еще один модуль, где для нашей мелодии будут определены соответствующие таблицы частот и длительностей нот, который поместит их адреса в регистры SI и BP, а затем вызовет процедуру PLAY. В примере 8.3 показан модуль мелодии "У Мэри была маленькая овечка". Шутки ради мы приводим в примере 8.4 таблицы данных для доброй старой мелодии "Индюк в соломе".

ПРИМЕР 8.2. ПРОЦЕДУРА ИСПОЛНЕНИЯ МЕЛОДИИ

```
; Эта процедура заставляет динамик ЭВМ "сыграть" мелодию, заданную
; двумя таблицами в сегменте данных
; Перед вызовом загрузите адрес таблицы частот в регистр SI и адрес
; таблицы длительностей в регистр BP
; Таблица частот должна заканчиваться значением 0FFFFH
; Эта процедура вызывает процедуру SOUND (из примера 8.1)
; Значения всех регистров сохраняются
;
EXTRN       SOUND:FAR            ;SOUND - внешняя процедура
PLAY        PROC  FAR
            PUBLIC  PLAY
            PUSH  BX             ;Сохранить значения регистров
            PUSH  DI
            PUSH  SI
            PUSH  BP
FREQ:       MOV   DI,[SI]        ;Извлечь значение частоты
            CMP   DI,0FFFFH      ;Конец мелодии?
            JE    END_PLAY       ; Да. Выйти из процедуры
            MOV   BX,DS:[BP]     ; Нет. Извлечь длительность
            CALL  SOUND          ;"Сыграть" мелодию
            ADD   SI,2           ;Продвинуть указатели таблиц
            ADD   BP,2
            JNZ   FREQ           ;Перейти к обработке следующей ноты
END_PLAY:   POP   BP             ;Восстановить значения регистров
            POP   SI
            POP   DI
            POP   BX
            RET                  ; и выйти из процедуры
PLAY        ENDP
```

ПРИМЕР 8.3. МОДУЛЬ ДАННЫХ, ТРЕБУЕМЫЙ ДЛЯ ИСПОЛНЕНИЯ МЕЛОДИИ

```
; Этот модуль определяет таблицы данных для мелодии "У Мэри была
; маленькая овечка" и помещает адрес таблицы частот в регистр SI,
; а таблицы длительностей - в регистр BP
;
; Ниже определяется сегмент стека
;
```

```
EXTRN           PLAY:FAR
MARY_STACK      SEGMENT  PARA STACK 'STACK'
                DB       64 DUP('STACK   ')
MARY_STACK      ENDS
;
; Ниже определяется сегмент данных
;
MARY_DATA       SEGMENT  PARA 'DATA'
MARY_FREQ       DW  330,294,262,294,3 DUP(330)        ;Такты 1-2
                DW  294,294,294,330,392,392           ;Такты 3-4
                DW  330,294,262,294,4 DUP(330)        ;Такты 5-6
                DW  294,294,330,294,262,0FFFFH        ;Такты 7-8
;
MARY_TIME       DW  6 DUP(25),50                      ;Такты 1-2
                DW  2 DUP(25,25,50)                   ;Такты 3-4
                DW  12 DUP(25),100                    ;Такты 5-8
MARY_DATA       ENDS
;
; Ниже следует сегмент команд
;
MARY_CODE       SEGMENT  PARA 'CODE'
MARY_PROC       PROC     FAR
                ASSUME   CS:MARY_CODE,DS:MARY_DATA,SS:MARY_STACK
;
; Занести в стек такие значения, чтобы эта процедура могла возвра-
; титься в DOS
;
                PUSH     DS          ;Поместить в стек номер блока адреса
;                                      возврата
                SUB      AX,AX       ;Обнулить регистр
                PUSH     AX          ;Поместить в стек нулевое смещение
;                                      адреса возврата
;
; Присвоить начальное значение адресу сегмента данных
;
                MOV      AX,SEG MARY_DATA
                MOV      DS,AX
;
; Присвоить начальные значения регистрам SI и BP и вызвать проце-
; дуру PLAY
;
                LEA      SI,MARY_FREQ
                LEA      BP,DS:MARY_TIME
                CALL     PLAY
                RET
MARY_PROC       ENDP
MARY_CODE       ENDS
                END      MARY_PROC
```

ПРИМЕР 8.4. ТАБЛИЦЫ ДАННЫХ ДЛЯ МЕЛОДИИ "ИНДЮК В СОЛОМЕ".

```
TRKY_FREQ  DW  294,262,233,233,262,233,175,147,156   ;Линейка 1
           DW  175,196,175,147,175,233,262,294,294   ;Линейка 2
           DW  294,262,233,262
           DW  294,262,262,294,262,233,233,262       ;Линейка 3
           DW  233,175,147,156
           DW  175,196,175,147,175,233,262,294,349   ;Линейка 4
           DW  349,294,233,262
           DW  294,262,233,294,349,294,349,349       ;Линейка 5
           DW  294,349,294,349,311,392,311,392,392   ;Линейка 6
           DW  311,392,311,392,466,466,349,349       ;Линейка 7
           DW  294,294,262,233,262,294,349,392       ;Линейка 8
           DW  349,294,233,262,294,262,233,0FFFFH
;
```

```
TRKY_TIME  DW  2 DUP(12),25,6 DUP(12)                    ;Линейка 1
           DW  4 DUP(12),25,2 DUP(12),2 DUP(25)          ;Линейка 2
           DW  4 DUP(12)
           DW  3 DUP(25),2 DUP(12),25,6 DUP(12)          ;Линейка 3
           DW  4 DUP(12),25,2 DUP(12),2 DUP(25)          ;Линейка 4
           DW  4 DUP(12)
           DW  2 DUP(25),50,12,25,12,2 DUP(25)           ;Линейка 5
           DW  12,25,12,50,2 DUP(12,25),25               ;Линейка 6
           DW  12,25,12,50,4 DUP(25)                     ;Линейка 7
           DW  3 DUP(25),3 DUP(12),25,5 DUP(12)          ;Линейка 8
           DW  3 DUP(25)
```

МУЗЫКА С КЛАВИАТУРЫ

Зная соответствующие нотам частоты, мы можем продвинуться на шаг дальше и разработать процедуру, заставляющую динамик исполнять ноту при наборе соответствующей ей буквы на клавиатуре ЭВМ[1].

Вряд ли можно найти приемлемый способ получения всех 88 нот клавиатуры пианино, но нам нетрудно добиться получения нот двух октав, изображенных на рис. 8.2. Для этого мы разработаем такую программу, с помощью которой при наборе буквы на верхнем регистре будет получена нота второй октавы.

А как быть с *диезами* и *бемолями*? На клавиатуре ЭВМ нет знака *бемоль*, но есть знак *диез* (#), который набирается нажатием клавиши с цифрой 3 на верхнем регистре. Поэтому мы предлагаем Вам набирать 3 для указания *диеза* и (совершенно произвольно) 2 для указания *бемоля*.

При сочинении музыки знаки *диез* и *бемоль* помещаются непосредственно за обозначением ноты (например, *ля-диез* записывается как А#). Поэтому идеальной для ЭВМ будет такая процедура, при которой исполнитель будет указывать ноту с *диезом*, сначала набирая букву, а затем цифру 3 (например А3). Однако при этом от ЭВМ потребуется после ввода очередной буквы ждать нажатия клавиши 2 (*бемоль*), 3 (*диез*) или какой-либо буквы. Чтобы избежать этого, мы потребуем от исполнителя указывать признак *диеза* или *бемоля* перед нажатием буквенной клавиши.

В примере 8.5 показана процедура KBD_MUSIC, которая позволяет Вам "нажимать" клавиши пианино с клавиатуры ЭВМ. Обратите внимание, что она содержит в сегменте данных таблицу частот для двух октав и вызывает процедуру SOUND из примера 8.1 для исполнения ноты в течение 0,1 с.

Во время работы с процедурой KBD_MUSIC набор буквы на основном регистре соответствует ноте первой октавы, а набор буквы на верхнем регистре – ноте второй октавы. Чтобы получить ноту с *бемолем* или *диезом*, перед ее набором нажмите клавиши 2 или 3 (на основном или верхнем регистре). Для выхода в операционную систему DOS нажмите клавишу возврата каретки.

ПРИМЕР 8.5. МУЗЫКА, ЗВУЧАЩАЯ С ПОМОЩЬЮ КЛАВИАТУРЫ

```
; Эта процедура заставляет клавиатуру ЭВМ выполнять функции клавиа-
; туры пианино с двумя октавами (рис. 8.2)
; Для исполнений основной ноты первой октавы нажмите клавишу с соот-
; ветствующей буквой (от A до G). Для исполнения основной ноты вто-
; рой октавы наберите букву на верхнем регистре. Для исполнения бе-
```

[1] В дальнейшем используются принятые за рубежом обозначения нот латинскими буквами: A, B, C, D, E, F, G соответствуют нотам *ля, си, до, ре, ми, фа, соль*. – *Прим. перев.*

```
; моля или диеза нажмите соответственно клавишу 2 или 3, а затем
; клавишу с буквой. Для завершения исполнения мелодии нажмите кла-
; вишу возврата каретки
; Эта процедура вызывает процедуру SOUND (из примера 8.1)
; Значения всех регистров сохраняются
;
            PAGE      ,132
EXTRN       SOUND:FAR            ;SOUND - внешняя процедура
KYBD_STACK       SEGMENT    PARA STACK 'STACK'
            DB   64 DUP('STACK    ')
KYBD_STACK       ENDS
KYBD_DATA        SEGMENT    PARA 'DATA'
            DW   220,247,131,147,165,175,196 ;Основные ноты первой октавы
            DW   440,494,262,294,330,349,392 ;Основные ноты второй октавы
            DW   233,262,139,156,175,185,208 ;Бемоли первой октавы
            DW   466,523,277,311,349,370,415 ;Бемоли второй октавы
            DW   208,233,123,136,156,165,185 ;Диезы первой октавы
            DW   415,466,247,277,311,330,370 ;Диезы второй октавы
KYBD_DATA        ENDS
;
; Ниже следует сегмент команд
;
KYBD_CODE        SEGMENT    PARA 'CODE'
KBD_MUSIC        PROC       FAR
                 ASSUME     CS:KYBD_CODE,DS:KYBD_DATA,SS:KYBD_STACK
;
; Занести в стек такие значения, чтобы эта процедура могла возвра-
; титься в DOS
;
                 PUSH     DS         ;Поместить в стек номер блока адреса
;                                     возврата
                 MOV      CX,0       ;Обнулить регистр
                 PUSH     CX
;
; Присвоить начальное значение адресу сегмента данных
;
                 MOV      AX,SEG KYBD_DATA
                 MOV      DS,AX
                 PUSH  AX            ;Сохранить значения регистров
                 PUSH  BX
                 PUSH  DI
                 PUSH  SI
NEW_NOTE:        MOV   AH,0          ;Считать очередную клавишу
                 INT   16H
                 CMP   AL,0DH        ;Нажата клавиша возврата каретки?
                 JNE   LWR_FLT
                 POP   SI            ; Да. Восстановить значения регистров
                 POP   DI
                 POP   BX
                 POP   AX
                 RET                 ; и выйти из процедуры
LWR_FLT:         SUB   SI,SI
                 CMP   AL,'2'
                 JNE   UPR_FLT
                 MOV   BX,56         ;Вероятно, диез первой октавы
GET_LC:          MOV   AH,0
                 INT   16H
                 JMP   SHORT  CHK_LO
UPR_FLT:         CMP   AL,'@'
                 JNE   LWR_SHP
                 MOV   BX,70         ;Вероятно, диез второй октавы
GET_UC:          MOV   AH,0
                 INT   16H
                 JMP   SHORT CHK_UO
LWR_SHP:         CMP   AL,'3'
                 JNE   UPR_SHP
                 MOV   BX,28         ;Вероятно, бемоль первой октавы
                 JMP   SHORT GET_UC
```

```
LWR_NAT:            MOV     BX,0            ;Вероятно, основная нота первой
;                                            октавы
                    CMP     AL,'a'
                    JB      UPR_NAT
                    CMP     AL,'g'
                    JA      NEW_NOTE
                    JNA     NEW_FREQ
UPR_NAT:            MOV     BX,14           ;Вероятно, нота второй октавы
                    JMP     SHORT CHK_UO
;
; Проверить, набрана ли нота первой октавы (a - g)
;
CHK_LO              CMP     AL,'a'
                    JB      NEW_NOTE
                    CMP     AL,'g'
                    JA      NEW_NOTE
                    JNA     NEW_FREQ
;
; Проверить, набрана ли нота второй октавы (A - G)
;
CHK_UO              CMP     AL,'A'
                    JB      NEW_NOTE
                    CMP     AL,'G'
                    JA      NEW_NOTE
;
; Преобразовать ASCII-код клавиши в индекс и загрузить его в регистр
; SI. Затем использовать его для извлечения значения частоты и вызвать
; процедуру SOUND для исполнения ноты в течение 0,1 с
;
NEW_FREQ:           AND     AL,0FH
                    SHL     AL,1
                    SUB     AH,AH
                    ADD     SI,AX
                    MOV     DI,FREQS-2[BX][SI]
                    MOV     BX,10               ;Длительность равна 0,1 с
                    CALL    SOUND
                    JMP     SHORT NEW_NOTE
KBD_MUSIC           ENDP
KYBD_CODE           ENDS
                    END     KBD_MUSIC
```

ГЛАВА 9. МАКРООПРЕДЕЛЕНИЯ

Макроопределения подобны подпрограммам и представляют собой "мини-программы", которые можно вставлять в исходные программы, просто указывая их имена. В этой главе мы расскажем, как создавать макроопределения и как пользоваться ими в программах. Затем мы представим набор полезных макроопределений и покажем, как образовать из них *библиотеку макроопределений*. Возможность работы с макроопределениями обеспечивается Макроассемблером фирмы IBM (MASM); в Малом Ассемблере не предусмотрено их использование.

9.1. ВВЕДЕНИЕ В МАКРООПРЕДЕЛЕНИЯ

Макроопределения представляют собой последовательность операторов на языке ассемблера (команд и псевдооператоров), которые могут несколько раз появиться в программе. Подобно процедурам макроопределения имеют имена. После того как макроопределение задано, его имя можно использовать в исходной программе вместо последовательности команд.

СРАВНЕНИЕ МАКРООПРЕДЕЛЕНИЙ И ПРОЦЕДУР

Хотя и макроопределения, и процедуры предоставляют возможность краткой ссылки на часто используемую последовательность команд, между ними существует и различие. Коды команд процедуры входят в программу однократно, и микропроцессор передает им управление (т. е. вызывает их командой CALL) по мере необходимости. Напротив, коды команд макроопределения могут встречаться в программе неоднократно; Ассемблер заменяет каждое упоминание имени макроопределения на те команды, которые оно представляет. (Другими словами, Ассемблер "расширяет" макроопределение.) Следовательно, при исполнении программы микропроцессор исполняет команды макроопределения "непосредственно", не передавая управление в другое место памяти (что имеет место при использовании процедурой). Таким образом, *имя макроопределения представляет собой директиву Ассемблера;* оно служит командой Ассемблеру, а не микропроцессору.

По сравнению с процедурами макроопределения имеют три преимущества:

1. Макроопределения *динамичны.* За счет изменения входных параметров макроопределения можно изменить не только объекты, которыми оно манипулирует, но и выполняемые над ними действия. Напротив, в случае процедуры можно изменять только передаваемые ей данные, что делает процедуры гораздо менее гибкими.
2. Применение макроопределений вместо процедур ускоряет исполнение программ, так как микропроцессору не надо отвлекаться на выполнение команд вызова процедуры и возврата из нее.
3. Макроопределения можно ввести в *библиотеку,* из которой программист может извлекать их при составлении других программ.

Однако ничто не дается даром. Макроопределения имеют основной недостаток, которого лишены процедуры: при их применении объектные программы становятся длиннее, поскольку макроопределения расширяются при каждом их появлении и память заполняется повторяющимися последовательностями команд.

МАКРООПРЕДЕЛЕНИЯ УСКОРЯЮТ ПРОГРАММИРОВАНИЕ

Макроопределения могут ускорить Вашу работу по программированию и отладке, а также модификацию программ, которая может понадобиться впоследствии. Они ускоряют Вашу работу по программированию потому, что достаточно один раз создать макроопределение, а затем им можно пользоваться в программе всюду, где оно требуется. Вместо задания определенной группы команд Вам достаточно указать только соответствующее макроопределение. (Тем самым макроопределения похожи на подпрограммы.)

Макроопределения могут ускорить Вашу работу по отладке, поскольку их можно создавать и отлаживать по отдельности. После того как макроопределение отлажено, Вам уже не надо задумываться над тем, правильно ли работает *эта* часть программы. Вы можете сосредоточиться на поиске ошибок в других ее частях.

Программы, включающие макроопределения, обычно легче читать и понимать. А это значит, что их легче и модифицировать. Предположим, например, что Вам требуется очистить экран, а затем перевести курсор в левый верхний угол. Для этого можно воспользоваться следующей последовательностью команд:

```
;
; Эти команды очищают экран
;
      MOV   CH,0       ;Начать со строки 0
      MOV   CL,0       ; и столбца 0
      MOV   DH,24      ;Закончить в строке 24
      MOV   DL,79      ; и столбце 79
      MOV   AH,6       ;Задать функцию прокрутки вверх
      MOV   AL,0       ;Очистить весь экран
      MOV   BH,7
      INT   10H        ;Инициировать прерывание типа 10
;
; Эти команды перемещают курсор в левый верхний угол экрана
;
      MOV   DH,0       ;Задать строку 0
      MOV   DL,0       ; и столбец 0
      MOV   AH,2       ;Задать функцию перемещения курсора
      INT   10H        ;Переместить курсор
```

Но если Вы зададите эти последовательности команд как макроопределения, то достаточно будет указать операторы

```
CLS          ;Очистить экран
HOME         ;Переместить курсор в исходную позицию
```

Как Вы считаете, какую из этих программ легче понять постороннему человеку? А какой из них Вы сами хотели бы заняться, скажем, спустя шесть месяцев?

Наконец, макроопределения делают модификацию программы более легкой. Стоит только изменить макроопределение и Ассемблер автоматически использует его новую версию всюду, где раньше использовалась старая.

СОСТАВ МАКРООПРЕДЕЛЕНИЙ

Каждое макроопределение имеет три части:

1. *Заголовок* – псевдооператор MACRO, в поле метки которого указано имя макроопределения, а в поле операнда – необязательный *список формальных параметров*. В списке формальных параметров указываются переменные – входные параметры, которые можно изменять при каждом вызове макроопределения.

2. *Тело* – последовательность операторов Ассемблера (команд и псевдооператоров), которые задают действия, выполняемые макроопределением.

3. *Концевик* – псевдооператор ENDM, который отмечает конец макроопределения. Если Вы опустите псевдооператор ENDM, то Ассемблер выдаст сообщение об ошибке End of file encountered on input file (обнаружен конец входного файла).

Ниже приведено простое макроопределение для сложения значений размером в слово:

```
ADD_WORDS  MACRO TERM1,TERM2,SUM
           MOV   AX,TERM1
           ADD   AX,TERM2
           MOV   SUM,AX
                 ENDM
```

Для Ассемблера безразлично, что будет указано в качестве операндов макроопределения; имена регистров, ячейки памяти или непосредственные значения (конечно, непосредственное значение нельзя использовать в качестве операнда SUM). Если окончательный результат имеет допустимую форму, то Ассемблер выполняет

подстановку операндов без лишних слов. Например, в одном месте программы можно сложить содержимое двух ячеек памяти с помощью оператора

```
ADD_WORDS PRICE,TAX,COST
```

Вместо этого оператора Ассемблер вставит в программу следующие команды:

```
MOV AX,PRICE
ADD AX,TAX
MOV COST,AX
```

А где-то в другом месте программы можно сложить содержимое двух регистров, указав оператор

```
ADD_WORDS BX,CX,DX
```

На этот раз Ассемблер вставит команды

```
MOV AX,BX
ADD AX,CX
MOV DX,AX
```

Обратите внимание, что гораздо легче передать параметры макроопределению, чем процедуре. В случае макроопределения Вы просто набираете имя параметра, а в случае процедуры должны переслать значение параметра в регистр или ячейку памяти.

9.2. ПСЕВДООПЕРАТОРЫ МАКРОАССЕМБЛЕРА

В табл. 9.1 описаны псевдооператоры, предусмотренные в Макроассемблере фирмы IBM для задания макроопределений. Они разделены на четыре группы: псевдооператоры общего назначения, псевдооператоры повторения, условные псевдооператоры и псевдооператоры управления листингом.

ПСЕВДООПЕРАТОРЫ ОБЩЕГО НАЗНАЧЕНИЯ

Мы уже обсуждали псевдооператор MACRO; он дает имя макроопределению и перечисляет имена его формальных параметров (если они имеются).

ПСЕВДООПЕРАТОР LOCAL

Если Ваше макроопределение содержит помеченные команды или псевдооператоры, то Вы должны указать Ассемблеру, чтобы он изменял метки при каждом расширении макроопределения. В противном случае Вы столкнетесь с сообщениями об ошибках Symbol is Multi-Defined (символ многократно определен). Псевдооператор LOCAL сообщает Ассемблеру, какие метки должны изменяться при каждом расширении.

Например, следующее ниже макроопределение WAIT заставляет микропроцессор ожидать, пока значение COUNT не уменьшится до нуля. Указание метки NEXT

Таблица 9.1. **Псевдооператоры Макроассемблера**

Псевдооператор	Назначение
Псевдооператоры общего назначения	
MACRO	Формат: **имя** MACRO [список формальных параметров] . . . . . . ENDM Присваивает **имя** последовательности операторов языка ассемблера Каждое определение MACRO должно завершаться псевдооператором ENDM
LOCAL	Формат: LOCAL [список формальных параметров] Заставляет Ассемблер создать уникальное имя для каждой метки из списка формальных параметров и подставить это имя при каждом вхождении метки в расширение макроопределения
Псевдооператоры повторения	
IRP	Формат: IRP **параметр,** <список аргументов> . . . . . . ENDM Заставляет Ассемблер повторять находящиеся между псевдооператорами IRP и ENDM операторы по одному разу для каждого аргумента списка. При каждом повторении производится подстановка очередного аргумента вместо каждого вхождения параметра в блок операторов
IRPC	Формат: IRPC **параметр, строка** . . . . . . ENDM Заставляет Ассемблер повторять находящиеся между псевдооператорами IRPC и ENDM операторы по одному разу для каждого символа строки. При каждом повторении производится подстановка очередного символа строки вместо каждого вхождения параметра в блок операторов
REPT	Формат: REPT **выражение** . . . . . . ENDM Заставляет Ассемблер повторять находящиеся между псевдооператорами REPT и ENDM операторы **выражение** раз

Псевдооператор	Назначение
Условные псевдооператоры	
EXITM	Формат EXITM Завершает расширение макроопределения в зависимости от результата выполнения условного псевдооператора
IF1	Формат: IF1 **выражение** ... ... ENDIF Выполняет, если Ассемблер осуществляет первый проход. Обычно используется для включения с помощью оператора INCLUDE файла с библиотекой макроопределения в исходную программу
IFB	Формат: IFB <**аргумент**> ... ... ENDIF Выполняется, если <**аргумент**> пуст. Угловые скобки обязательны.
IFNB	Формат: IFNB <**аргумент**> ... ... ENDIF Выполняется, если <**аргумент**> не пуст. Угловые скобки обязательны.
Псевдооператоры управления листингом	
.LALL	Формат: .LALL Вызывает выдачу полного листинга (включая комментарии) всех расширений макроопределения
.SALL	Формат: .SALL Исключает текст макроопределения из листинга
.XALL	Формат: .XALL Вызывает печать только тех строк макроопределения, которые генерируют объектный код. Этот режим устанавливается по умолчанию.

в операторе LOCAL позволяет нам пользоваться этим макроопределением в программе более одного раза:

```
WAIT   MACRO    COUNT
        LOCAL    NEXT
        PUSH     CX            ;Сохранить текущее значение CX
        MOV      CX,COUNT      ;Поместить счетчик в CX
NEXT:   LOOP     NEXT          ;Повторять, пока счетчик не обратится в 0
        POP      CX            ;Восстановить исходное значение CX
       ENDM
```

Обратите внимание на то,что оператор LOCAL следует непосредственно за оператором MACRO. Если требуется несколько операторов LOCAL, то *они должны быть первыми операторами в макроопределении.* Они должны предшествовать любому другому оператору и даже комментариям.

Указание метки в операторе LOCAL также означает, что Вы можете использовать такую же метку в других макроопределениях. Ассемблер дает ей новое внутреннее имя всякий раз, когда расширяет макроопределение; таким образом дублирование меток исключается.

ПСЕВДООПЕРАТОРЫ ПОВТОРЕНИЯ

Псевдооператоры REPT, IRP и IRPC заставляют Ассемблер повторить в макроопределении последовательность операторов на языке ассемблера.

Псевдооператор REPT получает свой счетчик числа повторений из *выражения,* указанного в поле операнда. Например, следующее макроопределение зарезервирует LENGTH байтов памяти и присвоит им в качестве начальных значений числа от 1 до LENGTH:

```
ALLOCATE  MACRO   TLABEL,LENGTH
TLABEL   EQU   THIS BYTE
VALUE   =   0
  REPT  LENGTH
  VALUE   =   VALUE+1
         DB   VALUE
  ENDM
          ENDM
```

Обратите внимание на то, что здесь нам понадобились два оператора ENDM: одним отмечен конец действия псевдооператора REPT, а другим – конец макроопределения.

После задания макроопределения ALLOCATE им можно воспользоваться для создания 40-байтовой таблицы TABLE1, указав такую последовательность операторов:

```
DATA       SEGMENT PARA  DATA
ALLOCATE   TABLE1,40
DATA       ENDS
```

Следующий псевдооператор повторения IRP позволяет Вам перечислить аргументы, которые должны быть подставлены вместо *формального параметра* при каждом повторении. Например, последовательность операторов

```
IRP VALUE,<1,2,3,5,7,11,13,17,19,23>
 DW VALUE*VALUE*VALUE
ENDM
```

создаст таблицу из 10 слов, содержащую кубы первых 10 простых чисел.

Псевдооператор IRPC похож на псевдооператор IRP, но его аргументами являются не числа, а строковые переменные. Ему требуются два операнда: *формальный параметр* и *строка символов:* он повторяет блок операторов для каждого символа строки. При каждом повторении очередной символ строки подставляется вместо каждого вхождения *формального параметра* в операторы блока.

Например, последовательность операторов

```
IRPC CHAR,0123456789
  DB CHAR
ENDM
```

создает в памяти 10-байтовую строку символов, содержащую ASCII-коды цифр от 0 до 9.

УСЛОВНЫЕ ПСЕВДООПЕРАТОРЫ

Эти псевдооператоры похожи на условные операторы, обсуждавшиеся в разд. 2.8. А именно, Ассемблер проверяет, удовлетворяется ли определенное условие. Если да, то он транслирует операторы, указанные между псевдооператорами IF и ENDIF; если нет, он пропускает их.

ПСЕВДООПЕРАТОР IF1

Псевдооператор IF1 (If Pass 1 – если первый проход) используется для считывания файла с библиотекой макроопределений в исходную программу. Мы продолжим его обсуждение в разд. 9.4.

ПСЕВДООПЕРАТОР IFB

Если Вы указали меньше параметров, чем это сделано при задании макроопределения, то Ассемблер присвоит опущенным (или пустым) параметрам нулевые значения (если только Вы не позаботитесь об этих замещениях сами). Оператор IFB (If Blank – если пустой) позволяет Вам указать альтернативные способы обработки пустых параметров. Обычно он используется, чтобы заставить макроопределения завершиться раньше в том случае, если какие-либо необходимые ему параметры отсутствуют. К упреждающему завершению мы еще вернемся при обсуждении в этом разделе оператора ENDM.

ПСЕВДООПЕРАТОР IFNB

Когда Ассемблер обнаруживает псевдооператор IFNB (If Not Blank – если не пуст), то он транслирует связанные с ним команды только в том случае, если пользователь дал значение параметру; в противном случае он пропускает их.

Например, макроопределение, считывающее фамилию, может быть рассчитано на то, что Вы будете вызывать его следующим образом:

```
GET_NAME  FIRST-NAME,MIDDLE-INITIAL,LAST-NAME
```

Макроопределение GET_NAME должно включать команды, которые получают имя, инициал отчества, а затем фамилию. Однако отчество не всегда известно, и

поэтому надо предусмотреть возможность его отсутствия. Вы можете сделать это с помощью псевдооператора IFNB, включая команды получения инициала отчества только в том случае, если пользователь задал соответствующий параметр. Следовательно, макроопределение GET_NAME должно иметь следующий общий вид:

```
GET_NAME  MACRO first-name,middle-initial,last-name
          . .    (Эти команды считывают имя)
          . .
IFNB     <middle-initial>
          . .    (Эти команды считывают инициал отчества)
          . .
ENDIF
          . .    (Эти команды считывают фамилию)
          . .
          ENDM
```

Приняв эти меры предосторожности, Вы можете воспользоваться формой оператора вида GET_NAME **Джон, Браун**, не задумываясь над тем, как Ассемблер прореагирует на отсутствующий инициал.

Псевдооператор IFNB может также помочь Вам избежать ошибок трансляции, возникающих из-за отсутствия операндов. Например, если Ваше макроопределение включает оператор PUSH reg_name и при его вызове Вы опустите в списке параметров параметр reg_name, то Ассемблер попытается оттранслировать этот оператор как PUSH 0, что, конечно, недопустимо. Чтобы избежать подобных ситуаций, используйте операторы

```
IFNB <reg_name>
  PUSH reg_name
ENDIF
```

(В этом случае Вам, вероятно, понадобится принять аналогичные предосторожности для последующего оператора POP reg_name.)

ПСЕВДООПЕРАТОР EXITM

Псевдооператор EXITM (exit Macro – выйти из макроопределения) заставляет Ассемблер завершить расширение макроопределения, не дожидаясь его конца, в зависимости от результата условного псевдооператора, например

```
IFB <name>
  EXITM
ENDIF
```

Вспомните макроопределение ALLOCATE, при задании которого мы использовали псевдооператор REPT. Ниже оно переопределено таким образом, чтобы Ассемблер резервировал место для таблицы только в том случае, если параметр LENGTH меньше 50:

```
ALL_LT_50  MACRO  LENGTH
      VALUE = 0
      IF LENGTH GE 50
        EXITM
      ENDIF
      REPT LENGTH
        VALUE = VALUE+1
        DB  VALUE
      ENDM
            ENDM
```

Псевдооператоры .LALL, .SALL и .XALL позволяют Вам управлять тем, какая часть текста макроопределения будет включена в листинговый файл, выдаваемый ассемблером (LST). Если Вы не указали ни один из них, то Ассемблер подразумевает режим листинга, задаваемый псевдооператором .XALL. Иначе говоря, в листинг войдут только те строки, для которых генерируется объектный код; самостоятельные комментарии и те псевдооператоры, для которых не резервируются ячейки памяти, будут опущены.

Псевдооператор .LALL вызывает выдачу полного листинга, включая комментарии, а псевдооператор .SALL исключает из листинга весь текст макроопределения. Вы можете использовать псевдооператор .LALL, чтобы получить копию программы для подшивки в архив, а затем оттранслировать ее с применением псевдооператора .SALL для получения листинга, которым Вы будете пользоваться после того, как макроопределения будут полностью отлажены.

9.3. ОПЕРАЦИИ В МАКРООПРЕДЕЛЕНИЯХ

Макроассемблер MACRO предоставляет возможность при задании макроопределений пользоваться четырьмя операциями (табл. 9.2).

ОПЕРАЦИЯ &

Операция & позволяет задавать модифицируемые метки и операнды. Например, следующее макроопределение образует таблицу байтов с заданным именем и заданной длиной:

```
DEF_TABLE MACRO SUFFIX,LENGTH
      TABLE&SUFFIX DB LENGTH DUP(?)
          ENDM
```

Таблица 9.2. **Операции в макроопределениях**

Операция	Назначение
&	Формат: **текст&текст** Вызывает конкатенацию (слияние) текста или имен
;;	Формат: ;; **комментарий** Исключает комментарий из листинга, даже если он выдается по команде .LALL
!	Формат: !**символ** Используется в аргументе для указания Ассемблеру, что **символ** надо использовать как литерал, а не как имя.
%	Формат: %**имя** Преобразует **имя** в число. При расширении макроопределения Ассемблер подставляет число вместо имени

Если в программе будет указан оператор DEF_TABLE A, 5, то Ассемблер превратит его в оператор

```
TABLEA DB 5 DUP(?)
```

ОПЕРАЦИЯ ;;

Операция ; ; заставляет Ассемблер опустить комментарии при расширении макроопределения. Без комментариев окончательная программа будет занимать меньше памяти и поэтому будет транслироваться быстрее. При задании макроопределений пользуйтесь обычным признаком комментария (;) только в абсолютно необходимых случаях и указывайте (; ;) в остальных случаях.

9.4. ЗАДАНИЕ МАКРООПРЕДЕЛЕНИЙ В ИСХОДНЫХ ПРОГРАММАХ

Существуют два способа использования макроопределений: их можно задавать в начале программы или считывать в программу из отдельного файла с библиотекой макроопределений. В этом разделе мы опишем, как задавать макроопределения непосредственно в программе. Библиотеки макроопределений будут обсуждаться в разд. 9.5.

Если Ваше макроопределение специфично и требуется только для одной программы, то его можно задать в тексте программы, а затем вызывать по мере необходимости. Обратимся к примерам.

В примере 7.1 была показана программа, изображавшая диагональную линию из "улыбающихся рожиц" по направлению к низу экрана. В примере 9.1 показана ее новая версия, которая использует макроопределения для сохранения значений регистров в стеке (PUSH_REGS) и восстановления значений регистров из стека (POP_REGS), для перемещения курсора (MOVE_CURSOR) и для изображения символа "рожицы" (PRINT_AL). Хотя из-за задания макроопределений текст программы в целом удлинился, зато процедурная часть программы несколько сократилась и стала более удобной для чтения.

Задание макроопределений непосредственно в тексте программы имеет недостаток – ими можно воспользоваться только в этой программе. Чтобы они были доступны и другим программам, их надо помещать в библиотеку макроопределений.

ПРИМЕР 9.1. ПРОГРАММА ИЗОБРАЖЕНИЯ ДИАГОНАЛЬНОЙ ЛИНИИ С ИСПОЛЬЗОВАНИЕМ МАКРООПРЕДЕЛЕНИЙ.

```
TITLE  Программа изображения диагонали с использованием макроопределений
;      (NEW_DIAG.ASM)
;
;***************************************************************************
;   Макроопределения
;***************************************************************************
PUSH_REGS  MACRO  reg_list
;;
;;  Сохранить значения регистров в стеке
;;
    IRP  reg,<reg_list>
         PUSH  reg
    ENDM
           ENDM
POP_REGS  MACRO  reg_list
;;
```

```
;;  Восстановить значения регистров из стека
;;
    IRP  reg,<reg list>
         POP   reg
    ENDM
          ENDM
MOVE_CURSOR  MACRO
;;
;;  Переместить курсор в строку и столбец, номера которых
;;  заданы соответственно в регистрах DH и DL
;;
         MOV     AH,2     ;;Задать функцию перемещения курсора
         INT     10H
              ENDM
PRINT_AL  MACRO
;;
;; Изобразить символ, заданный значением регистра AL
;;
         MOV     CX,1     ;;Изобразить ровно один символ
         MOV     AH,14    ;;Задать функцию вывода на экран
         INT     10H
          ENDM
;*****************************************************************************
;   Начало программы
;*****************************************************************************
STACK   SEGMENT  PARA STACK 'STACK'
       DB  64 DUP('STACK   ')
STACK  ENDS
OUR_CODE  SEGMENT  PARA 'CODE'
DIAG_LINE   PROC  FAR
           ASSUME  CS:OUR_CODE,SS:STACK
;
; Поместить в стек такие начальные значения, чтобы эта программа
; могла возвратить управление операционной системе DOS
;
          PUSH  DS          ;Поместить в стек номер блока адреса возврата
          MOV   DI,0        ;Обнулить регистр
          PUSH  DI          ;Поместить в стек нулевой адрес возврата
;
; Основная процедура
;
         PUSH_REGS  <AX,BX,CX,DX>  ;Сохранить значения регистров
         STI                 ;Разрешить обработку прерываний
         MOV     AH,15       ;Получить в BH номер активной страницы экрана
         INT     10H
         MOV     AH,0        ;Задать текстовый ч/б режим 80x25
         INT     10H
         MOV     DX,0        ;Начать со сттроки 0, столбца 0
SET_CRSR:   MOVE_CURSOR ;Задать новую позицию курсора
         MOV     AL,2        ;Изобразить "улыбающуюся рожицу"
         PRINT_AL
         INC     DH          ;Указать на следующую строку
         INC     DL          ; и новый столбец
         CMP     DH,25       ;Конец экрана?
         JNE     SET_CRSR
         POP_REGS  <DX,CX,BX,AX>       ;Да. Восстановить регистры
         RET                           ; и выйти из программы

DIAG_LINE   ENDP
OUR_CODE    ENDS
            END     DIAG_LINE
```

9.5. БИБЛИОТЕКА МАКРООПРЕДЕЛЕНИЙ

Библиотека макроопределений представляет собой дисковый файл, в котором заданы макроопределения, необходимые для нескольких программ. После того как такой файл создан, его содержимое можно считывать в любую

исходную программу. Тем самым все макроопределения библиотеки становятся доступными для этой программы. Чтобы использовать какое-либо из них, достаточно указать его имя.

СОЗДАНИЕ БИБЛИОТЕКИ МАКРООПРЕДЕЛЕНИЙ

Библиотека макроопределений представляет собой исходный файл, содержащий их текст. Следовательно, библиотечный файл можно создать с помощью программы EDLIN или программы обработки текстов точно так же, как Вы создаете обычную программу.

УКАЗАНИЯ ДЛЯ ЗАДАНИЯ МАКРООПРЕДЕЛЕНИЙ

Включаемые в библиотеку макроопределения должны быть процедурами общего назначения, которыми можно воспользоваться практически в любой программе. Поэтому Вы должны разрабатывать их так, чтобы они не только выполняли свое предназначение, но и не приводили к конфликтам в любой использующей их программе. Ниже приводится несколько указаний, которые могут помочь Вам разрабатывать эффективные макроопределения.

1. Документируйте макроопределения как можно тщательнее. Включайте побольше комментариев. Помните, смысл Ваших макроопределений должен быть понятен любому, кто будет ими пользоваться, а не только Вам.
2. При вводе комментариев пользуйтесь операцией ; ;, а для разделения полей оператора нажимайте клавиши табуляции (вместо ввода пробелов). Эти шаги помогут свести размеры Ваших программ к минимуму, что ускорит их трансляцию.
3. Старайтесь делать макроопределения как можно более универсальными. Если Вам требуется специфичное макроопределение, то по возможности выражайте его через более универсальные макроопределения. Например, представленная в этой главе библиотека макроопределений включает макроопределение LOCATE, которое перемещает курсор в заданную строку и заданный столбец. На его основе мы можем создать отдельное макроопределение HOME, перемещающее курсор в левый верхний угол экрана. Оно задается как LOCATE 0,0.
4. Если макроопределение содержит метки, перечислите их в операторе LOCAL.
5. Сохраняйте все используемые в макроопределении регистры, за исключением выходных. Как обычно, это делается с помощью операторов PUSH в начале макроопределения и POP в конце.
6. Если в макроопределении требуются действия, выполняемые ранее заданным макроопределением, то воспользуйтесь вызовом последнего.

СЧИТЫВАНИЕ БИБЛИОТЕКИ МАКРООПРЕДЕЛЕНИЙ В ПРОГРАММУ

Для считывания библиотеки макроопределений в исходную программу надо передать Ассемблеру ее имя в операторе INCLUDE. Однако если Вы сделаете это, скажем, оператором INCLUDE MACRO.LIB, то Ассемблер будет считывать библиотеку как во время прохода 1, так и во время прохода 2, что совсем не требуется.

Чтобы избежать повторения считывания, оператор INCLUDE надо поместить в *условную структуру* IF1, например

```
IF1
  INCLUDE MACRO.LIB
ENDIF
```

Это заставит Ассемблер считать библиотеку во время прохода 1. Но при этом указанный в операторе INCLUDE текст в листинг не попадет, поскольку Ассемблер выдает его во время прохода 2.

УДАЛЕНИЕ МАКРООПРЕДЕЛЕНИЙ

Пользование макробиблиотекой имеет тот недостаток, что при указании ее имени в операторе INCLUDE Ассемблер считывает все заданные в ней макроопределения, которые вовсе не требуются в данной программе; тем самым рабочая область заполняется ненужной информацией, что в принципе может способствовать возникновению ошибки out of memory (память исчерпана). Чтобы избежать этого, Вы можете *удалить* ненужные макроопределения и тем самым освободить занятую для них часть рабочей области. Для этого непосредственно за оператором INCLUDE надо перечислить имена подлежащих удалению макроопределений, например

```
INCLUDE MACRO.LIB
PURGE   MAC1,MAC2,MAC3
```

ОПИСАНИЕ МАКРООПРЕДЕЛЕНИЙ

В заключение этой главы мы опишем и дадим текст более 30 макроопределений, которые при желании Вы можете включить в библиотеку. Эти макроопределения выполняют обычные функции операционной системы DOS и языка Бейсик (например, очистку экрана, чтение данных с клавиатуры), а также ряд других функций, обычно требуемых при программировании на языке ассемблера (например, сохранение регистров, преобразование строк символов в числа и обратно).

В табл. 9.3 перечислены имена этих макроопределений, разделенных на семь функциональных групп: макроопределения общего назначения, макроопределения для преобразования данных, макроопределения для вывода изображения на

Таблица 9.3. **Состав библиотеки макроопределений**

Макроопределения общего назначения		
DELAY	POP_REGS	SET_ES
MAKE_STACK	RAND	
PUSH_REGS	SET_DS	
Макроопределения для преобразования данных		
$2BIN	BIN2$	KEY_$2BIN
Макроопределения для адресации курсора		
CLS	HOME	MOVE_CURSOR
CR	LF	POS
CRLF	LOCATE	TAB

Макроопределения для вывода изображений на экран		
MESSAGE	PRINT_NUMBER	SHOW_AL
MESSAGE_DX	PRINT$	VIDEO_STATE
PRINT_AL	PRINT_DX	
Макроопределения для ввода данных с клавиатуры		
INKEY	IN$	
INKEY_I	IN$_DX	
Макроопределения для работы со звуковым генератором		
BEEP	SOUND	SOUND_DI_BX
Макроопределение для работы с таймером		
READ_TIME	SET_TIME	

экран, макроопределения для ввода данных с клавиатуры, макроопределения для работы со звуковым генератором и макроопределения для работы с таймером. В следующих подразделах приводятся более детальные сведения.

МАКРООПРЕДЕЛЕНИЯ ОБЩЕГО НАЗНАЧЕНИЯ

DELAY [, минуты] [, секунды] [, сотые-доли-секунды]

Действие — Заставляет программу приостановиться на заданное время.
Входные значения — Могут быть указаны минуты (от 0 до 59), секунды (от 0 до 59) и сотые-доли-секунды (от 0 до 99)
Выходные значения — Отсутствуют

Примеры

```
DELAY ,30,50      ;Пауза в 30,5 с
DELAY 1,30        ;Пауза в 1,5 мин
```

MAKE_STACK имя-стека

Действие — Создает стандартный сегмент стека (т. е. резервирует 64 повторения строки 'STACK') и присваивает ему заданное имя.
Входные значения — **имя-стека** – имя сегмента стека.
Выходные значения — Отсутствуют.
Пример:

```
MAKE_STACK MY_STACK    ;Сегмент стека MY_STACK
```

PUSH_REGS <рег 0 [, рег 1, . . .]>

Действие — Сохраняет регистры в стеке в заданном порядке.
Входные значения — Окаймленный угловыми скобками список имен регистров, разделенных запятыми.
Выходные значения — Отсутствуют.
Пример:

```
PUSH_REGS <AX,BX,DX>    ;Сохранить AX, BX и DX
```

POP_REGS <рег 0 [, рег1, . . .]>

Действие	Извлекает из стека значения регистров в заданном порядке.
Входные значения	Окаймленный угловыми скобками список имен регистров, разделенных запятыми.
Выходные значения	Содержимое указанных регистров заменяется на значения из стека.
Примечание	Имена регистров перечисляются в обратном порядке по отношению к макроопределению PUSH_REGS.
Пример:	`POP_REGS <DX,BX,AX>    ;Восстановить DX, BX и AX`

RAND предел

Действие	Генерирует случайное число в интервале от 0 до **предел.**
Входные значения	**предел** (от 4 до 127)
Выходные значения	(AL) = случайное число
Пример:	`RAND 85    ;Выдать случайное число в интервале 0-85`

SET_DS имя-сегмента-данных

Действие	Загружает адрес сегмента данных в регистр DS.
Входные значения	**имя-сегмента-данных** – имя сегмента данных.
Выходные значения	Содержимое регистра DS указывает на сегмент данных

SET_ES имя-дополнительного-сегмента

Действие	Загружает адрес дополнительного сегмента в регистр ES.
Входные значения	**имя-дополнительного-сегмента** – имя дополнительного сегмента.
Выходные значения	Содержимое регистра ES указывает на дополнительный сегмент.

МАКРООПРЕДЕЛЕНИЯ ДЛЯ ПРЕОБРАЗОВАНИЯ ДАННЫХ

$2BIN имя строки

Действие	Преобразует строку символов, находящуюся в сегменте данных, в число со знаком, помещаемое в регистр AX.
Входные значения	**имя-строки** указывает на начальный адрес строки символов.
Формат строки	Строка может иметь длину до семи байтов; при этом первый байт служит счетчиком символов; первым значащим символом может быть знак "–" или "+", остальные должны быть цифрами от 0 до 9. Например `MY_STRING DB 6, -16540`
Выходные значения	Если строка символов допустима, то CF = 0 и (AX) = полученное число. Если строка символов недопустима, то CF = 1.
Примечание	Для преобразования числа, набираемого на клавиатуре, используйте макроопределение KEY_$2BIN.
Пример:	`$2BIN MY_STRING    ;Преобразовать MY_STRING в число` `JC PRINT_ERROR`

BIN2$ имя-строки

Действие	Преобразует содержимое регистра AX, рассматриваемое как число со знаком, в строку символов, находящуюся в сегменте данных.
Входные значения	**имя-строки** указывает на начальный адрес строки символов.
Формат строки	Семь зарезервированных байтов, например `MY_STRING DB 7 DUP(?)`
Выходные значения	Отсутствуют, регистр AX не изменяется.
Пример:	`BIN2$ MY_STRING ;Преобразовать содержимое AX в строку символов`

KEY_$2BIN

Действие	Преобразует последовательность набранных на клавиатуре символов в число со знаком, помещаемое в регистр AX.
Входные значения	Отсутствуют.
Выходные значения	Если последовательность символов допустима, то CF = 0 и (AX) = полученное число. Если недопустима, то CF = 1.

Пример:

```
KEY_$2BIN   ;Ввести число с клавиатуры
JC PRINT_ERROR
```

PRINT_NUMBER — См. макроопределения для вывода изображения на экран.

МАКРООПРЕДЕЛЕНИЯ ДЛЯ АДРЕСАЦИИ КУРСОРА

CLS

Действие	Очищает экран.
Входные значения	Отсутствуют.
Выходные значения	Отсутствуют.

CR

Действие	Возврат каретки; перемещает курсор к началу текущей строки.
Входные значения	Отсутствуют.
Выходные значения	Отсутствуют.
Примечание	См. также CRLF

CRLF

Действие	Возврат каретки и переход к следующей строке; перемещает курсор к началу следующей строки.
Входные значения	Отсутствуют.
Выходные значения	Отсутствуют.

HOME

Действие	Перемещает курсор в левый верхний угол экрана.
Входные значения	Отсутствуют.
Выходные значения	Отсутствуют.
Примечание	То же, что у команды LOCATE 0,0.

LF

Действие	Переход к новой строке; перемещает курсор на следующую строку.
Входные значения	Отсутствуют.
Выходные значения	Отсутствуют.
Примечание	1. См. также CRLF. 2. Если курсор уже находится на нижней строке экрана, то LF перемещает его на верхнюю строку.

LOCATE строка, столбец

Действие	Перемещает курсор в указанную позицию.
Входные значения	**строка** (от 0 до 24) и **столбец** (от 0 до 79).
Выходные значения	Отсутствуют.
Примечания	1. Аналогично макроопределению MOVE_CURSOR, только LOCATE использует не содержимое регистров, а значения пользователя. 2. Если **строка** превышает 24, то LOCATE перемещает курсор в строку 0. 3. Если **столбец** превышает 79, то LOCATE перемещает курсор в столбец 0. 4. См. также HOME.
Пример:	`LOCATE 24,79   ;Переместить курсор в правый нижний угол`

MOVE_CURSOR

Действие	Перемещает курсор в строку с номером, равным содержимому регистра DH, и в столбец с номером, равным содержимому регистра DL.
Входные значения	(DH) = номер строки; (DL) = номер столбца.
Выходные значения	Отсутствуют. Содержимое регистров DH и DL не изменяется.
Примечания	1. Аналогично макроопределению LOCATE, но MOVE_CURSOR использует содержимое регистров, а не значения пользователя. 2. Если содержимое регистра DH превышает 24, то курсор перемещается в строку 0. 3. Если содержимое регистра DL превышает 79, то курсор перемещается в столбец 0. 4. Для считывания текущей позиции курсора используйте макроопределение POS.

POS

Действие	Считывает текущую позицию курсора.
Входные значения	Отсутствуют.
Выходные значения	(DH, DL) = номера строки, столбца; (CH, CL) = режим изображения курсора.

TAB столбец

Действие	Перемещает курсор в указанный столбец.
Входные значения	**столбец** (от 0 до 79).

Выходные значения Отсутствуют.

Примечание Если курсор находится позади указанной позиции, то TAB переместит его в этот столбец следующей строки.

Пример:

```
TAB 40    ;Перейти к столбцу 40
```

МАКРООПРЕДЕЛЕНИЯ ДЛЯ ВЫВОДА ИЗОБРАЖЕНИЯ НА ЭКРАН

MESSAGE имя-строки

Действие Изображает на экране строку символов, находящуюся в сегменте данных.

Входные значения **имя-строки.**

Формат строки Строка должна заканчиваться символом $.
Например

```
ILLEGAL DB  Ошибочное входное значение. Повторите. $
```

Выходные значения Отсутствуют.

Примечание Аналогично макроопределению MESSAGE_DX, но MESSAGE берет указатель строки по имени, заданному пользователем, а не из регистра.

Пример:

```
MESSAGE ILLEGAL    ;Изобразить сообщение ILLEGAL
```

MESSAGE_DX

Действие Изображает на экране строку символов из сегмента данных, смещение которой содержится в регистре DX.

Входные значения (DS:DX) = начальный адрес строки символов.

Формат строки Тот же, что и для MESSAGE.

Выходные значения Отсутствуют; содержимое регистров DS и DX не изменяется.

Примечание Аналогично макроопределению MESSAGE, но MESSAGE_DX берет указатель строки из регистра, а не по имени, заданному пользователем.

Пример:

```
LEA DX,ILLEGAL    ;Изобразить сообщение ILLEGAL
MESSAGE_DX
```

PRINT_AL

Действие Изображает символ, код которого содержится в регистре AL, затем перемещает курсор в следующую позицию.

Входные значения (AL) = код символа.

Выходные значения Отсутствуют. Содержимое регистра AL не изменяется.

Примечание Чтобы курсор остался в текущей позиции, пользуйтесь макроопределением SHOW_AL.

Пример:

```
MOV AL,'P'    ;Изобразить P, затем переместить
PRINT_AL      ; курсор
```

PRINT_NUMBER

Действие Изображает число со знаком, содержащимся в регистре AX.

Входные значения (AX) = изображаемое число.

Выходные значения Отсутствуют; содержимое регистра AX не изменяется.

PRINT$ имя-строки

Действие Изображает на экране строку символов, находящуюся в сегменте данных.

Входные значения **имя-строки.**
Формат строки Первый байт строки должен быть счетчиком символов.
Выходные значения Отсутствуют.
Примечание Аналогично макроопределению PRINT$_DX, но PRINT$ получает свой указатель строки по имени, заданному пользователем, а не из регистра.

PRINT$_DX

Действие Изображает на экране строку символов из сегмента данных, смещение которой содержится в регистре DX.
Входные значения (DS:DX) = начальный адрес строки символов.
Формат строки Тот же, что и для PRINT$.
Выходные значения Отсутствуют; содержимое регистров DS и DX не изменяется.
Примечание Аналогично макроопределению PRINT$, но PRINT$_DX получает свой указатель строки из регистра, а не по имени, заданному пользователем.

SHOW_AL

Действие Изображает символ, код которого содержится в регистре AL, но не перемещает курсор.
Входные значения (AL) = код символа.
Выходные значения Отсутствуют; содержимое регистра AL не изменяется.
Примечание Если после изображения символа курсор надо переместить в следующую позицию, воспользуйтесь макроопределением PRINT_AL.
Пример:

```
MOV AL,'P'      ;Изобразить P в текущей позиции курсора
SHOW_AL
```

VIDEO_STATE

Действие Возвращает режим изображения.
Входные значения Отсутствуют.
Выходные значения (AL) = текущий режим;
(AH) = число столбцов на экране;
(BH) = номер активной страницы.

МАКРООПРЕДЕЛЕНИЯ ДЛЯ ВВОДА ДАННЫХ С КЛАВИАТУРЫ

INKEY

Действие Ожидает нажатия на клавишу, затем считывает ее ASCII-код в регистр AL и изображает соответствующий ему символ на экране.
Входные значения Отсутствуют.
Выходные значения (AL) = ASCII-код клавиши.
Примечания 1. Для считывания кода клавиши без изображения символа используйте макроопределение INKEY_I.
2. Таблицу ASCII-кодов см. в приложении Б.

INKEY_I

Действие Ожидает нажатия на клавишу, затем считывает ее ASCII-код в регистр AL, но символ на экране не изображает.

Входные значения Отсутствуют.
Выходные значения (AL) = ASCII-код клавиши.
Примечания 1. Для изображения считанного Вами символа используйте макроопределение INKEY.
2. Таблицу ASCII-кодов см. в приложении Б.

IN$ имя-строки

Действие Считывает последовательность набираемых на клавиатуре символов в буфер, находящийся в сегменте данных.
Входные значения **имя-строки** = имя буфера.
Формат буфера Буфер должен быть на три байта длиннее, чем помещаемая в него строка; первый байт определяет максимально возможную длину строки плюс один. Например, следующий буфер способен воспринять строки длиной до 10 символов:

```
STRING_BUFF DB 11,12 DUP(?)
```

Выходные значения Второй байт буфера содержит фактическую длину считанной строки (в байтах).
Примечания 1. Аналогично макроопределению IN$_DX, но IN$ получает свой указатель буфера от пользователя, а не из регистра.
2. Макроопределение IN$ считывает символы до тех пор, пока пользователь не нажмет клавишу ENTER. Если буфер переполняется, ЭВМ издает звуковой сигнал.
Пример:

```
IN$ STRING_BUFF   ;Считать символы в буфер
```

IN$_DX

Действие Считывает последовательность набираемых на клавиатуре символов в буфер, находящийся в сегменте данных.
Входные значения (DS:DX) = начальный адрес буфера.
Формат буфера Тот же, что и для IN$.
Выходные значения Второй байт буфера содержит фактическую длину считанной строки (в байтах); содержимое регистров DS и DX не изменяется.
Примечания 1. То же, что и IN$, но IN$_DX получает свой указатель буфера из регистра, а не от пользователя.
2. Макроопределение IN$_DX считывает символы до тех пор, пока пользователь не нажмет клавишу ENTER. Если буфер переполняется, ЭВМ издает звуковой сигнал.
Пример:

```
LEA DX,STRING_BUFF   ;Считать символы в буфер
IN$_DX
```

KEY_$2BIN (см. макроопределения для преобразования данных).

МАКРООПРЕДЕЛЕНИЯ ДЛЯ РАБОТЫ СО ЗВУКОВЫМ ГЕНЕРАТОРОМ

BEEP

Действие Издает звуковой сигнал.
Входные значения Отсутствуют.
Выходные значения Отсутствуют.
Примечание Аналогично макроопределению SOUND 1000,50.

SOUND частота, длительность

Действие	Издает тон заданной частоты и длительности.
Входные значения	**частота** в Гц (от 21 до 65 535) и **длительность** в сотых долях секунды (от 0 до 65 535).
Выходные значения	Отсутствуют.
Примечания	Аналогично макроопределению SOUND_DI_BX, но SOUND использует значения пользователя, а не регистры.
Пример:	`SOUND 262,100   ;"До" второй октавы в течение 1 с`

SOUND_DI_BX

Действие	Выдает тон, частота которого задана в регистре DI, а длительность – в регистре BX.
Входные значения	(DI) = частота в Гц (от 21 до 65 535). (BX) = длительность в сотых долях секунды (от 0 до 65 535).
Выходные значения	Отсутствуют; содержимое регистров DI и BX не изменяется.
Примечание	Аналогично макроопределению SOUND, но SOUND_DI_BX использует регистры, а не значения пользователя.
Пример:	see below

```
MOV  DI,262      ;"До" второй октавы в течение 1 с
MOV  BX,100
SOUND_DI_BX
```

МАКРООПРЕДЕЛЕНИЯ ДЛЯ РАБОТЫ С ТАЙМЕРОМ

READ_TIME

Действие	Чтение текущего времени.
Входные значения	Отсутствуют.
Выходные значения	(CH) = часы; (CL) = минуты; (DH) = секунды; (DL) = сотые доли секунды.

SET_TIME [часы] [, минуты] [,секунды] [,сотые-доли-секунды]

Действие	Установка на таймере заданного времени.
Входные значения	Могут быть заданы **часы** (от 0 до 23), **минуты** (от 0 до 59), **секунды** (от 0 до 59), и **сотые-доли-секунды** (от 0 до 99); опущенные операнды становятся нулевыми.
Выходные значения	Отсутствуют.
Пример:	`SET_TIME 1,,30    ;Задать время 1:00:30`

ТЕКСТ МАКРООПРЕДЕЛЕНИЙ

Заключительная часть этой главы содержит тексты каждого из макроопределений библиотеки в алфавитном порядке их имен. Если у Вас возникли предложения по улучшению какого-либо из этих макроопределений или желание добавить новые макроопределения, не откажите в любезности связаться с автором через фирму Brady Communicatious по адресу Bowie, ND 20 715.

Многие из макроопределений библиотеки сами вызывают другие макроопределения. Следовательно, если Вы измените или удалите макроопределения, это может затронуть другие макроопределения. Таблица 9.4 содержит пе-

Таблица 9.4. **Таблица перекрестных ссылок в библиотеке макроопределений**

Макроопределение	Используется	Использует
$2BIN	KEY_$2BIN	–
BEEP	–	SOUND
BIN2$	PRINT_NUMBER	–
CLS	–	–
CR	CRLF	MOVE_CURSOR, POS
CRLF	–	CR, LF
DELAY	–	READ_TIME
HOME	–	LOCATE
IN$	KEY_$2BIN	–
IN$_DX	IN$	–
INKEY	–	–
INKEY_I	–	–
KEY_$2BIN	–	$2BIN, IN$
LF	CRLF, TAB	MOVE_CURSOR, POS
LOCATE	HOME	MOVE_CURSOR
MAKE_STACK	–	–
MESSAGE	–	MESSAGE_DX
MESSAGE_DX	MESSAGE	–
MOVE_CURSOR	CR, LF, LOCATE	VIDEO_STATE
POP_REGS	–	–
POS	CR, LF, TAB	VIDEO_STATE
PRINT_AL	PRINT$_DX	VIDEO_STATE
PRINT_NUMBER	–	BIN2$, PRINT$
PRINT$	PRINT_NUMBER	PRINT$_DX
PRINT$_DX	PRINT$	–
PUSH_REGS	–	–
RAND	–	–
READ_TIME	DELAY	–
SET_DS	–	–
SET_ES	–	–
SET_TIME	–	–
SHOW_AL	–	VIDEO_STATE
SOUND	BEEP	SOUND_DI_BX
SOUND_DI_BX	SOUND	–
TAB	–	LF, POS, VIDEO_STATE
VIDEO_STATE	CR, LF, MOVE_CURSOR, POS, PRINT_AL, SHOW_AL, TAB	–

рекрестные ссылки для макроопределений библиотеки. В ней для каждого макроопределения указано, какие макроопределения оно использует и в каких макроопределениях используется само.

```
$2BIN MACRO string_name
    LOCAL BLANKS,CHK_NEG,CHK_POS,GO_CONV,GOOD,NO_GOOD,THRU
    LOCAL CONV_AB,RANGE,NON_DIG,DIGIT,END_CONV,SKIPIT
;;
;; Преобразует указанную строку в двоичное число со знаком,
;; загружаемое в регистр AX. Первый байт строки должен содер-
;; жать счетчик символов
;; Если преобразование проведено успешно, то CF = 0; в против-
;; ном случае CF = 1
;;
     PUSH     BX                        ;;Сохранить рабочие регистры
     PUCH     CX
     SUB      AX,AX                     ;;Начальное значение результата = 0
     SUB      CH,CH                     ;;Считать счетчик в CX
     MOV      CL,string_name
     LEA      BX,string_name+1          ;;Поместить первый адрес в BX
BLANKS:   CMP    BYTE PTR [BX],         ;;Пропустить ведущие пробелы
     JNE      CHK_NEG
     INC      BX
     LOOP     BLANKS
;;
;; Выполнить эти команды, если строка начинается со знака "минус"
;;
CHK_NEG:  CMP    BYTE PTR [BX], -
     JNE     CHK_POS
     INC     BX                    ;;Продвинуть указатель
     DEC     CX                    ;;Уменьшить счетчик
     CAL     CONV_AB               ;;Преобразовать строку
     JC      THRU
     CMP     AX,32768              ;;Число слишком мало?
     JA      NO_GOOD
     NEG     AX                    ;; Нет. Обратить знак результата
     JS      GOOD
;;
;; Выполнить эти команды, если первый символ строки не "минус"
;;
CHK_POS:  CMP  BYTE PTR [BX],'+'
    JNE      GO_CONV
    INC      BX                    ;;Продвинуть указатель
    DEC      CX                    ;;Уменьшить счетчик
GO_CONV:  CALL CONV_AB             ;;Преобразовать строку
     JC      THRU
     CMP     AX,32767              ;;Число слишком велико?
     JA      NO_GOOD
GOOD:  CLC
     JNC     THRU
NO_GOOD:  STC                      ;; Да. Установить флаг переноса
THRU:  POP   CX                    ;;Восстановить значения регистров
     POP     BX
     JMP     SKIPIT
;;
;; Следующая процедура выполняет фактическое преобразование
;;
CONV_AB  PROC
     PUSH    BP                    ;;Сохранить значения рабочих регистров
     PUSH    BX
     PUSH    SI
     MOV     BP,BX                 ;;Поместить указатель в регистр BP
     SUB     BX,BX                 ;; и обнулить регистр BX
RANGE:   CMP  BYTE PTR DS:[BP],'0'  ;;Если символ не является
     JB      NON_DIG                   ;; цифрой,
     CMP     BYTE PTR DS:[BP],'9'
     JBE     DIGIT
```

```
NON_DIG:  STC                              ;; то установить флаг переноса
    JC      END_CONV                       ;; и выйти из процедуры
DIGIT:   MOV  SI,10                   ;;Так как символ - цифра,
     PUSH    DX
     MUL     SI                       ;; то умножить AX на 10
     POP     DX
     MOV     BL,DS:[BP]               ;;Извлечь ASCII-код,
     AND     BX,0FH                   ;; оставить только младшие биты
     ADD     AX,BX                    ;; и модифицировать результат
     JC      END_CONV                 ;;Выйти из процедуры, если результат
;;                                         слишком велик
     INC     BP                       ;;В противном случае увеличить BP
     LOOP    RANGE                    ;; и продолжить
     CLC                              ;;Все сделано. Обнулить флаг переноса
END_CONV:  POP  SI                    ;;Восстановить значения регистров
     POP     BX
     POP     BP
     RET
CONV_AB   ENDP
SKIPIT:   NOP
     ENDM

BEEP MACRO

;;
;; Заставить динамик сигналить в течение 1/2 с на частоте 1000 Гц
;;
     SOUND  1000,50
      ENDM

BIN2$ MACRO string_name

     LOCAL    FILL_BUFF,CLR_DVD,NO_MORE
;;
;; Преобразовать число со знаком, находящееся в регистре AX, в строку
;; сегмента данных. Строка должна иметь длину семь байтов. Первый байт
;; в качестве значения получит счетчик символов
;;
     PUSH     DX              ;;Сохранить значения используемых регистров
     PUSH     CX
     PUSH     BX
     PUSH     SI
     PUSH     AX
     LEA      BX,string_name+1    ;;BX укажет на первый символ
     MOV      CX,6                ;;Заполнить буфер пробелами
FILL_BUFF:    MOV  BYTE PTR [BX],
     INC      BX
     LOOP     FILL_BUFF
     MOV      SI,10               ;;Приготовиться к делению на 10
     OR       AX,AX               ;;Если значение отрицательное,
     JNS      CLR_DIV
     NEG      AX                  ;; то изменить его знак
CLR_DVD:  SUB  DX,DX              ;;Обнулить старшую половину делимого
     DIV      SI                  ;;Поделить AX на 10
     ADD      DX,'0'              ;;Преобразовать остаток в ASCII-цифру
     DEC      BX                  ;;Попятиться в буфере
     MOV      [BX],DL             ;;Поместить символ в строку
     OR       AX,AX               ;;Все сделано?
     JNZ      CLR_DVD             ;; Нет. Взять следующую цифру
     POP      AX                  ;; Да. Взять исходное значение
     OR       AX,AX               ;;Оно отрицательное?
     JNS      NO_MORE
     DEC      BX                  ;; Да. Поместить в строку знак
     MOV      BYTE PTR [BX],'-'
NO_MORE:  MOV  string_name,6      ;;Зафиксировать счетчик символов
     POP      SI                  ;;Восстановить значения регистров
     POP      BX
     POP      CX
     POP      DX
        ENDM
```

```
CLS MACRO

;;
;; Стереть экран
;;
     PUSH AX               ;;Сохранить значения используемых регистров
     PUSH BX
     PUSH CX
     PUSH DX
     MOV  CX,0             ;;Начать со строки 0, столбца 0
     MOV  DH,24            ;;Закончить в строке 24
     MOV  DL,79            ;; и столбце 79
     MOV  AH,6             ;;Задать режим прокрутки вверх
     MOV  AL,0             ;;Стереть весь экран
     MOV  BH,7
     INT  10H              ;;Инициировать прерывание типа 10
     POP  DX               ;;Восстановить значения регистров
     POP  CX
     POP  BX
     POP  AX
     ENDM

CR MACRO
;;
;; Переместить курсор к началу текущей строки
;;
     PUSH CX
     PUSH DX
     POS                   ;;Считать номер строки в регистр DH
     SUB  DL,DL            ;;Обнулить номер столбца
     MOVE CURSOR           ;;Переместить курсор
     POP  DX
     POP  CX
    ENDM
CRLF NACRO

;;
;; Переместить курсор к началу следующей строки
;;
     CR                    ;;Возврат каретки
     LF                    ;;Переход к следующей строке
      ENDM

DELAY MACRO minutes,seconds,hundredths

     LOCAL    SECS,MINS,HRS,CHECK,QUIT
;;
;; Выждать заданное время
;;
     PUSH    AX
     PUSH    BX
     PUSH    CX
     PUSH    DX
     READ TIME             ;;Считать текущее время
     MOV     AH,CH         ;;Скопировать часы в регистр AH
     MOV     AL,CL         ;;            минуты в AL
     MOV     BH,DH         ;;            секунды в BH
     MOV     BL,DL         ;;            сотые доли секунды в BL
;;
;; Получить время конца паузы, добавляя значения входных параметров
;;  к текущему времени
;;
     ADD     AL,minutes
     ADD     BH,seconds
     ADD     BL,hundred
;;
;; Учесть возможные пере
;;
```

```
      CMP     BL,100
      JB      SECS
      SUB     BL,100
      INC     BH
SECS:  CMP    BH,60
      JB      MINS
      SUB     BH,60
      INC     AL
MINS:  CMP    AL,60
      JB      HRS
      SUB     AL,60
      INC     AH
HRS:   CMP    AH,24
      JNE     CHECK
      SUB     AH,AH
;;
;; Ждать до наступления полученного времени
;;
CHECK:  READ_TIME        ;;Снова считать время
      CMP      CX,AX
      JA       QUIT
      JB       CHECK
      CMP      DX,BX
      JB       CHECK
 QUIT:  POP    DX
       POP     CX
       POP     BX
       POP     AX
         ENDM

 HOME MACRO

 ;;
 ;; Переместить курсор в левый верхний угол
 ;;
       LOCATE  0,0
        ENDM

 IN$   MACRO  string_name

 ;;
 ;; Считать набираемые на клавиатуре символы в заданный буфер,
 ;; находящийся в сегменте данных
 ;;
       PUSH     DX
       LEA      DX,string_name
       IN$_DX
       POP      DX
        ENDM

 IN$_DX  MACRO

 ;;
 ;; Считать набираемые на клавиатуре символы в буфер, находящийся
 ;; в сегменте данных. Адрес буфера = (DS:DX)
 ;;
       PUSH     AX
       MOV      AH,0AH
       INT      21H
       POP      AX
            ENDM

 INKEY  MACRO

 ;;
 ;; Считать с клавиатуры в регистр AL очередной символ и изобразить
 ;; его на экране
 ;;
```

```
        PUSH      CX             ;;Сохранить значения регистров
        PUSH      AX
        MOV       AH,1           ;;Выбрать режим чтения символа
        INT       21H            ;;Считать символ
        POP       CX             ;;Восстановить значение регистра AH
        POP       AH,CH
        POP       CX             ;;Восстановить значение регистра CX
          ENDM
INKEY_I   MACRO

;;
;; Считать с клавиатуры в регистр AL очередной символ, но не
;; изображать его
;;
        PUSH      CX             ;;Сохранить значения регистров
        PUSH      AX
        MOV       AH,8           ;;Выбрать режим чтения символа
        INT       21H            ;;Считать символ
        POP       CX             ;;Восстановить значение регистра AH
        POP       AH,CH
        POP       CX             ;;Восстановить значение регистра CX
          ENDM

KEY_$2BIN   MACRO

        LOCAL      TEMP,SAVE
;;
;; Считать с клавиатуры последовательность символов и преобразовать
;; ее в число со знаком, возвращаемое в регистре AX
;;
        JMP        SAVE          ;;Пропустить рабочий буфер
TEMP   DB     7,8 DUP(?)
SAVE:  PUSH     DS               ;;Сохранить значения используемых регистров
        PUSH       CX
        MOV        CX,CS         ;;Сделать DS указателем на сегмент команд
        MOV        DS,CX
        IN$        TEMP          ;;Считать строку в буфер TEMP
        $2BIN      TEMP+1        ;;Выполнить преобразование
        POP        CX            ;;Восстановить значения регистров
        POP        DS
               ENDM

LF   MACRO

;;
;; Переместить курсор на следующую строку (в ту же позицию столбца)
;; Если он находился на последней строке экрана, переместить его в
;; верхнюю строку
;;
        PUSH       CX
        PUSH       DX
        POS                      ;;Считать номер строки в регистр DH
        INC        DH            ;; и увеличить его на 1
        MOVE_CURSOR              ;;Переместить курсор
        POP        DX
        POP        CX
      ENDM

LOCATE   MACRO   row,col

;;
;; Переместить курсор в заданные строку и столбец
;;
        PUSH       DX
        MOV        DH,row
        MOV        DL,col
        MOVE_CURSOR
        POP        DX
           ENDM
```

```
MAKE_STACK   MACRO   stack_name

;;
;; Определить "стандартный" сегмент стека
;;
stack_name SEGMENT    PARA STACK   STACK'
           DB    64   DUP('STACK     ')
stack_name ENDS
           ENDM

MESSAGE   MACRO   string_name

;;
;; Изобразить строку с сообщением, находящуюся в сегменте данных. Строка
;; должна заканчиваться знаком $
;;
     PUSH      DX
     LEA       DX,DS:string_name
     MESSAGE_DX
         ENDM

MESSAGE_DX   MACRO

;;
;; Изобразить строку с сообщением, находящуюся в сегменте данных
;; Смещение адреса строки берется из регистра DX. Строка должна
;; заканчиваться знаком $
;;
     PUSH      AX
     MOV       AH,9          ;;Выбрать режим изображения строки
     INT       21H           ;;Изобразить строку
     POP       AX
               ENDM

MOVE_CURSOR   MACRO

     LOCAL     OK,VS
;;
;; Переместить курсор в строку и столбец, номера которых заданы в
;; регистрах DH и DL соответственно. Если номер строки превышает 24
;; или номер столбца превышает 79, то сделать его равным 0
;;
     PUSH      AX            ;;Сохранить значения используемых регистров
     PUSH      BX
     CMP       DH,24         ;;Если необходимо, прокрутить входные пара-
     JNA       OK            ;; метры
     SUB       DH,DH
OK:  CMP       DL,79
     JNA       VS
     SUB       DL,DL
VS:  VIDEO_STATE             ;;Считать номер страницы экрана в регистр BH
     MOV       AH,2          ;;Выбрать режим перемещения курсора
     INT       10H           ;;Переместить курсор
     POP       BX            ;;Восстановить значения регистров
     POP       AX
                ENDM

POP_REGS   MACRO   reg_list

;;
;; Извлечь из стека значения регистров
;;
     IRP    reg,<reg list>
     POP      reg
     ENDM
            ENDM
```

```
POS  MACRO

;;
;; Считать номера текущей строки и текущего столбца в регистры DH
;; и DL соответственно, а режим курсора - в регистры CH и CL
;;
     PUSH   AX              ;;Сохранить значения используемых регистров
     PUSH   BX
     VIDEO_STATE            ;;Считать номер страницы экрана в регистр BH
     MOV    AH,3            ;;Выбрать режим чтения позиции курсора
     INT    10H             ;;Считать его позицию
     POP    BX              ;;Восстановить значения регистров
     POP    AX
     ENDM

PRINT_AL  MACRO

;;
;; Изобразить символ, содержащийся в регистре AL, и продвинуть курсор
;; в следующую позицию
;;
     PUSH   AX              ;;Сохранить значения используемых регистров
     PUSH   BX
     PUSH   CX
     PUSH   AX              ;;Сохранить входное значение
     VIDEO_STATE            ;;Считать номер страницы экрана в регистр BH
     POP    AX              ;;Восстановить входное значение
     MOV    CX,1            ;;Изобразить ровно один символ
     MOV    AH,14           ;;Выбрать режим изображения
     INT    10H             ;;Изобразить символ
     POP    CX              ;;Восстановить значения регистров
     POP    BX
     POP    AX
         ENDM

PRINT_NUMBER  MACRO

     LOCAL  TEMP,SAVE
;;
;; Изобразить содержимое регистра AX как число со знаком
;;
     JMP    SAVE            ;;Обойти рабочий буфер
TEMP  DB  7 DUP(?)
SAVE:  PUSH  DS             ;;Сохранить значения используемых регистров
     PUSH   CX
     MOV    CX,CS           ;;Сделать DS указателем на сегмент команд
     MOV    DS,CX
     BIN2$  TEMP            ;;Выполнить преобразование
     PRINT$ TEMP            ;;Изобразить результат
     POP    CX              ;;Восстановить значения регистров
     POP    DS
           ENDM

PRINT$  MACRO  string_name

;;
;; Изобразить заданную строку, находящуюся в сегменте данных
;; Первый байт строки должен содержать счетчик символов
;;
     PUSH   DX
     LEA    DX,string_name
     PRINT$_DX
     POP    DX
        ENDM
```

```
PRINT$_DX   MACRO

      LOCAL    STRIP,NEXT_C,NO_BLNK
;;
;; Изобразить заданную строку, находящуюся в сегменте данных
;; Смещение адреса строки берется из регистра DX. Первый байт
;; строки должен содержать счетчик символов
;;
      PUSH    AX              ;;Сохранить значения используемых регистров
      PUSH    CX
      PUSH    SI
      SUB     CX,CX           ;;Обнулить регистр счетчика
      MOV     SI,DX           ;;Поместить указатель строки в регистр SI
      MOV     CL,[SI]         ;;Считать счетчик символов
      INC     SI              ;;Указать на первый символ строки
STRIP:   LODSB                ;;Пропустить ведущие пробелы
      CMP     AL,
      JNE     NO_BLNK
      LOOP    STRIP
NEXT_C:  LODSB                ;;Считать символ
NO_BLNK:  PRINT AL            ;; и изобразить его на экране
      LOOP    NEXT_C
      POP     SI              ;;Восстановить значения регистров
      POP     CX
      POP     AX
            ENDM

PUSH_REGS   MACRO   reg_list

;;
;; Поместить значения регистров в стек
;;
      IRP    reg,<reg_list>
      PUSH     reg
      ENDM
            ENDM

RAND   MACRO   limit

      LOCAL    STRIP
;;
;; Произвести случайное число в интервале 0..limit (включительно)
;; и возвратить его через регистр AL
;;
      PUSH     CX             ;;Сохранить значения используемых регистров
      PUSH     DX
      PUSH     AX
      MOV      AH,0           ;;Считать показания таймера
      INT      1AH
      MOV      AX,DX          ;;Поместить младший байт в регистр AX
      MOV      CL,limit       ;;Поместить limit в регистр CL
;;
;; Удалить из делимого (AX) достаточное число старших битов, чтобы
;; гарантировать отсутствие переполнения
;;
      MOV      DH,3FH         ;;Поместить в DH маску для операции AND
      CMP      CL,64
      JAE      STRIP
      SHR      DH,1           ;;Если limit < 64, удалить 3 бита
      CMP      CL,32
      JAE      STRIP
      SHR      DH,1           ;;Если limit < 32, удалить 4 бита
      CMP      CL,16
      JAE      STRIP
      SHR      DH,1           ;;Если limit < 16, удалить 5 битов
      CMP      CL,0
      JAE      STRIP
      SHR      DH,1           ;;Если limit < 8, удалить 6 битов
```

```
STRIP:   AND  AH,DH       ;;Удалить биты
     DIV      CL          ;;Разделить результат в AX на limit в CL
     MOV      AL,AH       ;;Поместить остаток в регистр AL
     POP      CX          ;;Восстановить значение регистра AH
     MOV      AH,CH
     POP      DX          ;;Восстановить значения остальных регистров
     POP      CX
      ENDM

READ_TIME  MACRO

;;
;; Считать текущее время в регистры CH (часы), CL (минуты), DH
;; (секунды) и DL (сотые доли секунды)
;;
     PUSH     AX
     MOV      AH,2CH      ;;Выбрать режим чтения времени
     INT      21H         ;;Считать время
     POP      AX
            ENDM

SET_DS  MACRO  dseg_name

;;
;; Загрузить адрес сегмента данных в регистр DS
;;
     PUSH     AX
     MOV      AX,dseg_name
     MOV      DS,AX
     POP      AX
        ENDM

SET_ES  MACRO. eseg_name

;;
;; Загрузить адрес дополнительного сегмента в регистр ES
;;
     PUSH     AX
     MOV      AX,eseg_name
     MOV      ES,AX
     POP      AX
        ENDM

SET_TIME  MACRO hours,minutes,seconds,hsecs

;;
;; Установить показания таймера. Пропущенные операнды считать нулями
;;
     PUSH    AX           ;;Сохранить значения используемых регистров
     PUSH    CX
     PUSH    DX
     MOV     CH,hours     ;;Считать значения пользователя
     MOV     CL,minutes
     MOV     DH,seconds
     MOV     DL,hsecs
     MOV     AH,2DH       ;;Выбрать режим установки таймера
     INT     21H          ;;Установить показания таймера
     POP     DX           ;;Восстановить значения регистров
     POP     CX
     POP     AX
         ENDM

SHOW_AL  MACRO

;;
;; Изобразить на экране символ, находящийся в регистре AL. Положение
;; курсора оставить без изменения
;;
```

```
        PUSH    AX              ;;Сохранить значения используемых регистров
        PUSH    BX
        PUSH    CX
        PUSH    AX              ;;Сохранить входное значение
        VIDEO_STATE             ;;Считать номер страницы экрана в регистр BH
        POP     AX              ;;Восстановить входное значение
        MOV     CX,1            ;;Изобразить ровно один символ
        MOV     AH,10           ;;Выбрать режим изображения
        INT     10H             ;;Изобразить символ
        POP     CX              ;;Восстановить значения регистров
        POP     BX
        POP     AX
            ENDM
SOUND   MACRO   freq,duration

;;
;; Вызвать звук с заданной частотой и длительностью. Частота
;; задается в Герцах, длительность - в сотых долях секунды
;;
        MOV     DI,freq
        MOV     BX,duration
        SOUND_DI_BX
          ENDM

SOUND_DI_BX  MACRO

;;
;; Вызвать звук с заданной частотой и длительностью. Частота
;; (в Герцах) берется из регистра DI, а длительность (в сотых
;; долях секунды) - из регистра BX
;;
        PUSH    AX              ;;Сохранить значения используемых регистров
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    DI
        MOV     AL,0B6H         ;;Записать в регистр режима таймера
        OUT     43H,AL
        MOV     DX,14H          ;;Делитель времени =
        MOV     AX,4F38H        ;; 1331000/частота
        DIV     DI
        OUT     42H,AL          ;;Записать младший байт счетчика таймера 2
        MOV     AL,AH
        OUT     42H,AL          ;;Записать старший байт счетчика таймера 2
        IN      AL,61H          ;;Считать текущий режим порта B
        MOV     AH,AL           ;; и сохранить его в регистре AH
        OR      AL,3            ;;Включить динамик
        OUT     61H,AL
WAIT:   MOV     CX,2801         ;;Выждать 10 мс
SPKR_ON:  LOOP  SPKR_ON
        DEC     BX              ;;Счетчик длительности звучания исчерпан?
        JNZ     WAIT            ;; Нет. Продолжить звучание
        MOV     AL,AH           ;;Восстановить режим порта
        OUT     61H,AL
        POP     DI              ;;Восстановить значения регистров
        POP     DX
        POP     CX
        POP     BX
        POP     AX
              ENDM
TAB   MACRO  column

        LOCAL   DO_TAB,GET_BH
;;
;; Переместить курсор в столбец с заданным номером (от 0 до 79)
;; Если эта позиция позади курсора, то переместить его в эту
;; позицию следующей строки
;;
```

```
        PUSH    AX              ;;Сохранить значения используемых регистров
        PUSH    BX
        PUSH    CX
        PUSH    DX
        POS                     ;;Считать номер столбца в регистр DL
        CMP     DL,column       ;;Эта позиция позади курсора?
        JBE     DO_TAB
        LF                      ;; Да. Переместить его на следующую строку
        POS                     ;; и считать новое положение курсора
DO_TAB:  MOV  DL,column         ;;Считать заданный номер столбца
        CMP     DL,79           ;;Он слишком велик?
        JNA     GET_BH
        MOV     DL,79           ;; Да. Заменить на столбец 79
GET_BH:  VIDEO_STATE            ;;Считать номер активной страницы в регистр BH
        MOV     AH,2            ;;Выбрать режим перемещения курсора
        INT     10H             ;;Переместить курсор
        POP     DX              ;;Восстановить значения регистров
        POP     CX
        POP     BX
        POP     AX
        ENDM

VIDEO_STATE  MACRO

;;
;; Считать текущий режим в регистр AL, число столбцов на экране в
;; регистр AH, а номер активной страницы - в регистр BH
;;
        MOV     AH,15           ;;Выбрать режим получения состояния изображения
        INT     10H             ;;Считать состояние изображения
                ENDM
```

ГЛАВА 10. БИБЛИОТЕКИ ОБЪЕКТНЫХ МОДУЛЕЙ

В гл. 9 мы описали, как создавать *библиотеки макроопределений,* т. е. дисковые файлы, содержащие их текст. Чтобы воспользоваться каким-либо макроопределением, входящим в состав библиотеки, надо включить ее в программу с помощью оператора INCLUDE, а затем указать имя макроопределения. Следовательно, библиотеки макроопределений позволяют Вам избавиться от необходимости набирать текст макроопределений в каждой программе, где они требуются.

Макроассемблер версии 2 предусматривает аналогичные возможности для программных модулей. А именно он позволяет Вам создавать *библиотеки объектных модулей,* т. е. дисковые файлы, содержащие объектные модули. Чтобы воспользоваться процедурой, входящей в состав библиотеки объектных модулей, надо вызвать ее в Вашей программе с помощью оператора CALL и, как обычно, объявить ее внешней (с помощью оператора EXTRN). Затем надо вызвать загрузчик и тем самым связать библиотеку с вызывающим процедуру программным модулем. Загрузчик автоматически извлечет вызываемую процедуру, как если бы она была отдельным модулем. Следовательно, библиотека объектных модулей позволяет Вам не беспокоиться о том, на каком диске находится процедура, и избежать постоянной смены дисков. Вам достаточно помнить только о том диске, на котором находится объектная библиотека.

10.1. СОСТАВЛЕНИЕ БИБЛИОТЕКИ ОБЪЕКТНЫХ МОДУЛЕЙ

На диске Ассемблера находится библиотекарь – программа LIB.EXE, манипулирующая библиотеками объектных модулей. Для создания библиотеки объектных модулей Вы должны сообщить библиотекарю ее имя и указать первый объектный модуль, который необходимо поместить в библиотеку. Чтобы это выполнить, надо поместить диск Ассемблера в дисковод A, а диск с объектным модулем – в дисковод B, а затем, как обычно, включить ЭВМ. Получите на экране приглашение к вводу B >, а затем введите команду вида

```
a:lib имя_библиотеки+объектный_модуль;
```

Библиотекарь автоматически добавит к именам библиотеки и объектного модуля расширения LIB и OBJ.

Например, для создания библиотеки OBJECT.LIB из модуля SORT.OBJ в качестве первого ее элемента надо ввести команду

```
a:lib object+sort;
```

10.2. ВЫПОЛНЕНИЕ ОПЕРАЦИЙ НАД БИБЛИОТЕКАМИ ОБЪЕКТНЫХ МОДУЛЕЙ

После того как библиотека создана, Вы можете добавлять к ней новые модули, повторяя предыдущую команду. Например, чтобы добавить модуль MULU32.OBJ к библиотеке OBJECT, надо ввести команду

```
a:lib object+mulu32;
```

Знак "+" означает здесь "добавить к библиотеке". Для удаления модуля из библиотеки вместо знака "+" используйте знак "–". Например, для удаления модуля MULU32.OBJ надо ввести команду

```
a:lib object-mulu32;
```

Вам может потребоваться скопировать модуль, входящий в состав библиотеки, например, чтобы добавить его к другой библиотеке. Для этого используется операция *. Так, чтобы сделать копию модуля SORT.OBJ, надо ввести команду

```
a:lib object*sort;
```

После ее выполнения в библиотеке по-прежнему будет находиться модуль SORT, а на диске появится его копия с именем SORT.OBJ.

Аналогично операция – * удалит объектный модуль из библиотеки, но при этом создаст его копию на диске. Например, команда

```
a:lib object-*sort;
```

переместит модуль SORT из библиотеки OBJECT в новый файл с именем SORT.OBJ.

Если Вы не помните, из каких объектных модулей состоит библиотека, то можете воспользоваться библиотекарем для получения файла с ее каталогом. Для этого надо ввести команду вида

```
a:lib имя_библиотеки,имя_библиотеки.dir;
```

А для изображения каталога на экране надо ввести команду

```
type имя_библиотеки.dir
```

Наряду с именами объектных модулей каталог содержит перечень тех имен каждого модуля, которые указаны в операторе PUBLIC. Это показывает Вам, какие имена должны использоваться в Вашей программе при ссылках на процедуры, переменные и константы, определенные в вызываемых ею модулях библиотеки.

10.3. ПОЛЬЗОВАНИЕ БИБЛИОТЕКАМИ ОБЪЕКТНЫХ МОДУЛЕЙ

Как упоминалось в начале этой главы, при пользовании библиотекой объектных модулей Вы должны *связать* с ней объектный модуль (или модули), содержащий Вашу программу. Эта задача выполняется загрузчиком, который изображает на экране характерное приглашение к вводу. Загрузчик вызывается командой вида

```
a:link module,,NUL
```

Когда ЭВМ выдаст приглашение к вводу

```
Libraries [.LIB]:
```

введите имя Вашей библиотеки, а затем нажмите на клавишу возврата каретки. Например, чтобы связать библиотеку OBJECT с объектным модулем MAIN_PROG, надо ввести команду a:link main_prog, , nul, а затем ответить

```
Libraries [.LIB]: object
```

Если в Вашей программе требуются объектные модули из нескольких различных библиотек, то все их надо перечислить в ответ на приглашение Libraries. Например

```
Libraries [.LIB]: lib1+lib2+lib3
```

ГЛАВА 11. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

Одним из самых ценных качеств Макроассемблера версии 2 является то, что он позволяет Вам писать *структурированные* программы на языке ассемблера. Под "структурированными" мы понимаем программы, содержащие операторы высокого уровня и выполняющие циклы, условные передачи управления и другие задачи управления, которые при отсутствии оператора потребовали бы не-

скольких команд на языке ассемблера. В действительности структурные операторы обеспечивают те же виды сложных операций, что и язык Бейсик. Например, одно из семейств структурных операторов (или *логических структур*) выполняет те же действия, что и оператор IF-THEN-ELSE в языке Бейсик, а другое семейство имеет то же назначение, что и операторы FOR-NEXT.

Но структурное программирование вовсе не сводится к простой замене команд на структурные операторы. Оно, скорее, представляет собой *философию* разработки программного обеспечения, в основу которой положены следующие элементы:

Разработка по методу "сверху вниз". (Напомним, что по этому методу разработка программ начинается с комментариев, к которым последовательными шагами добавляются все новые и новые детали.)
Программирование без меток.
Программирование без оператора GOTO (т. е. без команд условной или безусловной передачи управления).
Запись текста программы с отступами, показывающими уровень вложенности логических структур.
В достаточной мере самодокументирующиеся листинги.

Все это предназначено для повышения *качества* программ на языке ассемблера. Качественные улучшения должны быть явными в следущих аспектах:

разработчик должен лучше понимать логику программы;
листинг должен быть понятен тем, кто пытается внести собственные изменения в чужие программы;
поскольку программа более понятна, в ней должно быть меньше ошибок;
улучшается гибкость и облегчается сопровождение программы. Это означает, что программа окажется более устойчивой к изменению внешних ситуаций, чем это предусмотрено в проекте, и последующие модификации органично войдут в существующую структуру программы и не внесут новых ошибок;
Программа должна быть удобной для чтения, это поможет сделать листинг лучшей документацией программы.

Возможность пользования структурными операторами обеспечивается программой SALUT (Structured Assembly Language Utility – сервисная программа структурного программирования на языке ассемблера), которая находится на диске Макроассемблера версии 2. Вы должны набрать текст своей программы, а затем вызвать программу SALUT для преобразования структурных операторов в стандартные команды на языке ассемблера. После этого надо как всегда оттранслировать программу и загрузить ее.

В этой главе описаны структурные операторы программы SALUT и приведены примеры и указания по их применению. В ней также обсуждаются те шаги, которые надо выполнить для трансляции и форматирования структурированных программ.

11.1. СТРУКТУРНЫЕ ОПЕРАТОРЫ И СТРУКТУРЫ ЛОГИКИ УПРАВЛЕНИЯ

Существует больше дюжины различных структурных операторов, но ими надо пользоваться в строго определенных сочетаниях, которые в руководстве по Макроассемблеру называются *структурами логики управления*. В этом руководстве определены восемь различных структур логики управления, но мы можем разбить их на три группы:

Структура IF (если), реализующая принятие решений. Она заставляет микропроцессор 8088 исполнять или пропускать группу команд в зависимости от того, удовлетворяется заданное условие или нет. Структура IF аналогична оператору IF-THEN в языке Бейсик.

Структура DO (выполнить), обеспечивающая повторение группы команд. Она заставляет микропроцессор 8088 циклически выполнять одну или несколько групп команд, пока не удовлетворится заданное условие. Структура DO аналогична операторам FOR-NEXT в языке Бейсик.

Структура SEARCH (искать) также связана с повторением команд, но используется в том случае, когда Вам необходимо знать, какое именно из нескольких условий привело к завершению выполнения цикла. Структура SEARCH исполняет свой блок команд для каждого условия выхода из цикла.

УСЛОВИЯ В СТРУКТУРАХ ЛОГИКИ УПРАВЛЕНИЯ

В табл. 11.1 показаны условия, которыми Вы можете воспользоваться для управления структурами IF, DO и SEARCH. За исключением условия NCXZ, это те же самые условия, что используются в командах условной передачи управления (Jx), описанных в гл. 3.

Будьте осторожны: условия CXZ и NCXZ не всегда допустимы в структурах. Мы обсудим это детальнее в разд. 11.5.

11.2. СТРУКТУРА IF

Структура IF заставляет микропроцессор 8088 выполнять либо пропускать группу команд в зависимости от того, истинно или ложно заданное условие. Ее основная форма

```
$IF условие
    A
$ENDIF
```

где **условие** – проверяемое условие (одно из тех, что перечислены в табл. 11.1), а А – блок команд. Если **условие** истинно, то микропроцессор 8088 исполняет эти команды, в противном случае он переходит к оператору, следующему непосредственно за оператором $ENDIF. (Обратите внимание, что этот порядок действий противоположен тому, которому следуют команды условной передачи управления: последние *пропускают* команды, если условие истинно.) На рис. 11.1, *а* показана блок-схема структуры IF.

Например, для преобразования находящегося в регистре BX числа в его абсолютное значение Вы должны воспользоваться операторами

```
CMP BX,0        ;Отрицательное значение?
$IF L
    NEG BX      ;Если да, обратить знак
$ENDIF
```

СТРУКТУРА IF С ЧАСТИЦЕЙ ELSE

Как и в языке Бейсик, Вы можете вставить частицу ELSE и заставить микропроцессор 8088 исполнять альтернативную группу команд в случае, если условие ложно. Тогда стуктура IF примет вид

```
$IF условие
      A            (исполняется, если условие истинно)
$ELSE
      B            (исполняется, если условие ложно)
$ENDIF
```

На рис. 11.1, *б* показана блок-схема, поясняющая действие структуры $IF-$ELSE-$ENDIF.

Таблица 11.1. **Условия в структурах логики управления**

Условие	Описание	Истинно, если...
A	Above — выше	CF = 0 и ZF = 0
AE	Above or Egual — выше или равно	CF = 0
B	Below — ниже	CF = 1
BE	Below or Egual — ниже или равно	CF = 1 или ZF = 1
C	Carry — перенос	CF = 1
CXZ	CX Register is Zero — регистр CX равен нулю	(CX) = 0
E	Egual — равно	ZF = 1
* G	Greater — больше	ZF = 0 и SF = OF
* GE	Greater or Egual — больше или равно	ZF = OF
* L	Less — меньше	SF <> OF
* LE	Less or Egual — меньше или равно	ZF = 1 или SF <> OF
NA	Not Above — не выше	CF = 1 или ZF = 1
NAE	Not Above nor Egual — ни выше, ни равно	CF = 1
NB	Not Below — не ниже	CF = 0
NBE	Not Below nor Egual — ни ниже, ни равно	CF = 0 и ZF = 0
NC	No Carry — нет переноса	CF = 0
NCXZ	CX register Not Zero — регистр CX не равен нулю	(CX) <> 0
NE	Not Egual — не равно	ZF = 0
* NG	Not Greater — не больше	ZF = 1 или SF <> OF
* NGE	Not Greater nor Egual — ни больше, ни равно	SF <> OF
* NL	Not Less — не меньше	SF = OF
* NLE	Not Less nor Egual — ни меньше, ни равно	ZF = 0 и SF = OF
* NO	No Overflow — нет переполнения	OF = 0
NP	No Parity (Odd) — несовпадение четности (нечет)	PF = 0
* NS	No Sign — нет знака	SF = 0
NZ	Not Zero — не нуль	ZF = 0
* O	Overflow — переполнение	OF = 1
P	Parity (Even) — совпадение четности (чет)	PF = 1
PE	Parity Even — чет	PF = 1
PO	Parity Odd — нечет	PF = 0
* S	Sign — знак	SF = 1
Z	Zero — нуль	ZF = 1

* Используется при арифметических операциях над числами со знаком (в дополнительном коде).

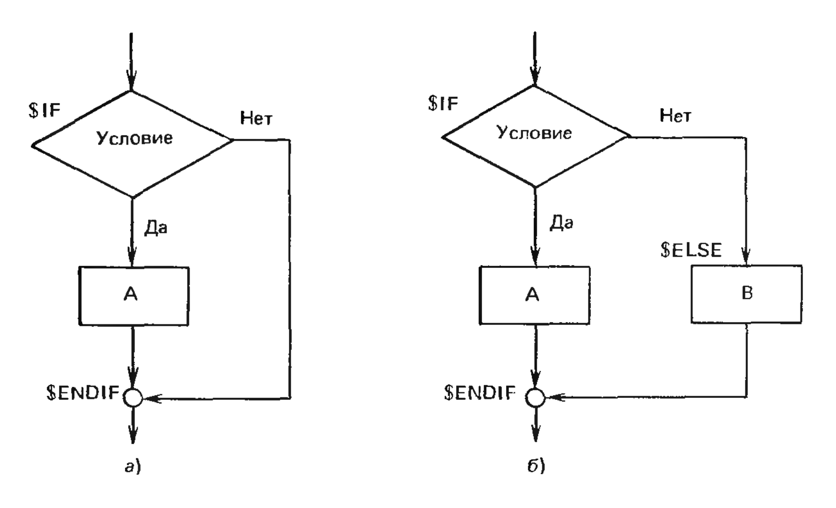


Рис. 11.1. Структуры IF (*а*), IF ELSE (*б*)

ФУНКЦИОНИРОВАНИЕ СТРУКТУРЫ IF

На самом деле операторы $IF, $ELSE и $ENDIF являются командами для программы SALUT. Когда Вы вызываете программу SALUT, то она преобразует эти операторы в строки комментариев (так что Ассемблер будет их игнорировать), а затем создает их эквивалент на языке ассемблера. После этого программа может быть оттранслирована Ассемблером (даже его малой версией), так как операторов $IF, $ELSE и $ENDIF в ней уже нет. В следующих разделах мы покажем, как программа SALUT преобразует сначала структуру $IF-$ENDIF, а затем структуру $IF-$ELSE-$ENDIF.

КОМАНДЫ, ГЕНЕРИРУЕМЫЕ ОПЕРАТОРАМИ $IF-$ENDIF

Когда программа SALUT обнаруживает оператор $IF, она генерирует команду условной передачи управления Jx, которая заставляет микропроцессор 8088 пропустить операторы, находящиеся между оператором $IF и последующим оператором $ENDIF. Обратите внимание на слово "пропустить". Оно сообщает Вам, что программа SALUT должна подставить команду Jx, *обратную* по отношению к смыслу операнда в операторе $IF. Следовательно, она должна заменить $IF Z на JNZ, а $IF A на JNA. Конечно, у каждой команды условной передачи управления должен быть адресат, и поэтому программа SALUT вставит метку после оператора $ENDIF. Например после того, как программа SALUT обработает операторы $IF и $ENDIF, наш предыдущий пример примет следующий вид:

```
  CMP BX,0       ;Отрицательное значение?
 ;$IF L
  JNL $$IF1
      NEG BX     ;Если да, обратить знак
 ;$ENDIF
$$IF1:
```

(Мы вставили метку $$IF1 только для иллюстрации, так как не знаем, да этого и не требуется, какую метку программа SALUT вставит на самом деле.)

При обработке структуры $IF-$ELSE-$ENDIF программа SALUT заменяет оператор $IF на команду Jx, передающую управление метке, следующей за $ELSE, заменяет $ENDIF на другую метку и подставляет вместо $ELSE команду *близкой* передачи управления JMP, адресующуюся к метке при $ENDIF. Таким образом, как показано на следующем схематическом листинге, программа SALUT преобразует операторы левой части в те, что указаны справа:

```
$IF условие                 JN условие $$IF1
  A                           A
$ELSE                       JMP SHORT $$EN1
                      $$IF1:
  B                           B
$ENDIF               $$EN1:
```

ВАРИАНТЫ ОПЕРАНДА

Общая форма операторов $IF и $ELSE

```
$IF   условие [,AND/OR][,LONG]
$ELSE   [LONG]
```

где операции AND и OR позволяют Вам указывать дополнительные условия, а атрибут LONG обеспечивает возможность использования длинных блоков команд в этой структуре.

ОПЕРАЦИИ AND И OR

До сих пор мы обсуждали структуры IF, зависящие от выполнения только одного условия, но (как и в языке Бейсик) Вы можете добиться того, чтобы в структуре IF рассматривалась комбинация условий в сочетании с операциями AND и OR.

Однако между условиями в языке Бейсик и условиями на языке ассемблера есть определенные различия. С одной стороны, микропроцессор рассматривает каждый член выражения в порядке его появления; здесь нет скобок, задающих порядок действий. С другой стороны, для каждого применения операции AND или OR требуется отдельный оператор $IF, так что между ними можно помещать команды.

Ниже приводится общий вид оператора $IF с двумя условиями, соединенными операцией IF:

```
      ...       (Вычислить условие1)
      ...
$IF   условие1,AND
      ...       (Вычислить условие2)
      ...
$IF   условие2
      ...       (Оба условия выполнены)
      ...
$ENDIF
```

Учтите, что команды вычисления **условие 2** выполняются только в том случае, если **условие 1** истинно. Если оно ложно, то микропроцессор пропустит все операторы вплоть до оператора $ENDIF. (Обратите внимание и на то, что мы использова-

ли только один оператор $ENDIF, поскольку операция AND объединяет операторы $IF в один *составной* оператор.)

С помощью команд, следующих за первым оператором $IF, обеспечивается установка флага (или флагов) для второго оператора $IF. Например, для выдачи сообщения "Вес в норме", если значение WEIGHT находится между 31 и 34, надо использовать операторы

```
        CMP WEIGHT,31
  $IF AE,AND
        CMP WEIGHT,34.
  $IF BE
        (выдать сообщение "Вес в норме")
  $ENDIF
```

Конечно, Вы можете включить сюда и частицу ELSE, чтобы выполнить определенные действия (например, выдачу сообщения "Вес за пределами нормы"), если какое-либо из условий не выполнено.

Ниже приводится общий вид оператора $IF, в котором условия связаны с операцией OR:

```
        ... (Вычислить условие1)
        ...
  $IF   условие1,OR
        ... (Вычислить условие2)
        ...
  $IF   условие2
        ... (условие1 или условие2 выполнено)
        ...
  $ENDIF
```

Здесь микропроцессор выполнит команды вычисления **условие2** только в том случае, если **условие1** оказалось невыполненным.

ДАЛЬНИЕ ПЕРЕДАЧИ УПРАВЛЕНИЯ

Так как программа SALUT заменяет операторы $IF и $ELSE на команды Jx и JMP типа SHORT (т.е. команды ближней передачи управления), то размер блока команд не может превышать 127 байт, т.е. около 50-60 команд, что обычно не является существенным ограничением. Однако в случае, если Ваша структура IF включает много команд, то надо приписать операторам $IF или $ELSE атрибут LONG.

Например, можно указать

```
  $IF A,LONG
```

Атрибут LONG заставляет программу SALUT заменить оператор $IF A на команды

```
        JA   $$XL1
        JMP  $$IF1
  $$XL1:
```

вместо команды JNA $$IF1.

11.3. СТРУКТУРА DO

Структура DO заставляет микропроцессор 8088 повторять блоки команд до тех пор, пока не выполнится заданное условие. Существуют следующие три ее формы:

Структура DO UNTIL (выполнять ... до) повторяет блок команд до тех пор, пока условие в конце этого блока не станет истинным.

Структура DO WHILE (пока не ..., выполнять) повторяет блок команд, пока условие в начале этого блока не станет истинным. (Другими словами, она выполняет блок, пока условие ложно.) Если изначальное условие истинно, то она пропускает блок, ни разу его не выполнив.

Структура DO COMPLEX (выполнять комплексно) представляет собой комбинацию структур DO UNTIL и DO WHILE, включающую в себя два блока команд. Она выполняет первый блок, затем проверяет условие. Если оно истинно, то выполняется второй блок и процесс повторяется заново; в противном случае второй блок пропускается и цикл завершается.

Примечание. Слова UNTIL (до), WHILE (пока) и COMPLEX (комплексно) использованы лишь для того, чтобы помочь различить эти три формы структуры DO; *не набирайте* их в текстах своих программ. (Правда, фирма IBM *предусматривала* альтернативную форму структуры DO COMPLEX, в которой слово COMPLEX появлялось и в программе, но мы здесь не будем ее обсуждать.)

СТРУКТУРА DO UNTIL

Эта структура имеет общий вид

```
$DO
   A
$ENDDO условие
```

где **условие** – это условие прекращения цикла. Так как микропроцессор проверяет **условие** в конце цикла, то команды блока A всегда выполняются не менее одного раза. На рис. 11.2,*a* показана блок-схема, описывающая действие структуры DO UNTIL.

Структура DO UNTIL часто используется для повторения команд в зависимости от значения счетчика в регистре CX. Например

```
MOV CX,10
$DO
     ... (Эти команды будут повторяться 10 раз)
     ...
$ENDDO LOOP
```

Здесь команда LOOP уменьшает содержимое регистра CX на единицу и возвращает управление оператору $DO, если значение регистра CX отлично от нуля. Эквивалентом на языке Бейсик служат операторы

```
FOR X=10 TO 1 STEP -1
...
...
NEXT X
```

СТРУКТУРА DO WHILE

Эта структура имеет общий вид

```
$DO
$LEAVE  условие
   A
$ENDDO
```

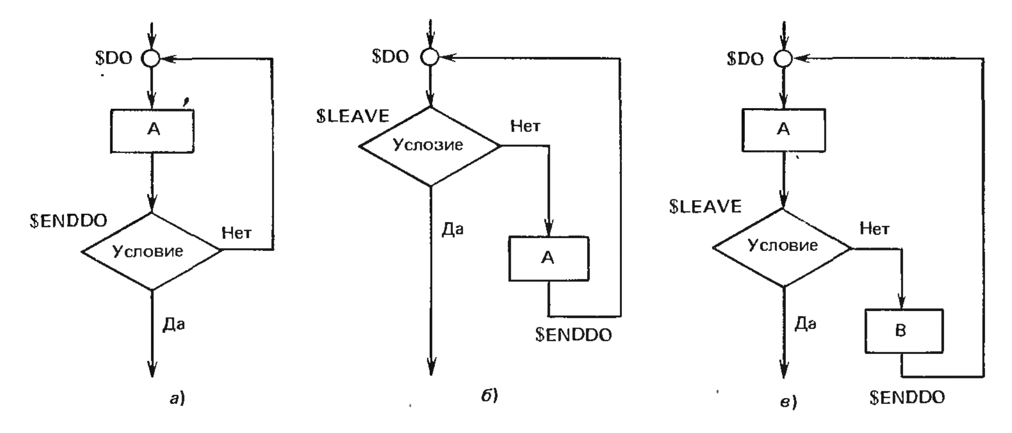


Рис. 11.2. Структуры DO UNTIL (*а*), DO WHILE (*б*), DO COMPLEX (*в*)

где **условие** – это условие завершения цикла. На рис. 11.2,*б* показаны блок-схема структуры DO WHILE.

Обратите внимание на то, что в структуре DO WHILE **условие** проверяется перед командами цикла, а не после них, как это происходит в структуре DO UNTIL. Следовательно, если **условие** изначально истинно, то микропроцессор обходит цикл, не исполнив его ни разу. Структура DO WHILE как бы сигнализирует микропроцессору: "Выполняй последующие команды, пока **условие** не выполняется".

Вам следует пользоваться структурой DO WHILE для обхода цикла, организованного с помощью счетчика в регистре CX, если регистр CX уже содержит нуль, например

```
$DO
$LEAVE CXZ
    ...
    ...
$ENDDO LOOP
```

СТРУКТУРА DO COMPLEX

Структура DO COMPLEX похожа на структуру DO WHILE, но позволяет Вам выполнить некоторые действия как до проверки в операторе LEAVE, так и после нее. Тем самым обеспечивается выход из середины цикла. Эта структура имеет следующий общий вид:

```
$DO
  A
$LEAVE   условие
  B
$ENDDO
```

Как и раньше, **условие** представляет собой условие завершения цикла. На рис. 11.2,*в* показана блок-схема структуры DO COMPLEX.

Здесь блок команд A подготавливает **условие**, а блок команд B выполняет то, что требуется проделать, если **условие** ложно. Например, для приостанова прог-

раммы до тех пор, пока пользователь не введет правильный пароль, можно воспользоваться следующим приемом:

```
$DO
   (Выдайте на экран "Пожалуйста, введите Ваш пароль:")
   (Проверьте пароль. Если он правилен, положите флаг CF
     равным 1.)
$LEAVE CF
   (Выдайте на экран "Вы ошиблись. Попробуйте еще раз".)
$ENDDO
```

ДОПОЛНИТЕЛЬНЫЕ ОПЕРАНДЫ

Подобно оператору $IF, описанному в разд. 11.2, операторы $ENDDO и $LEAVE допускают указание операции AND (И) или OR (ИЛИ), что позволяет Вам комбинировать условия. Кроме того, в них можно указывать атрибут LONG, обеспечивающий возможность пользоваться длинными блоками команд. Наконец, в операторе $ENDDO можно указывать параметр LOOP, что избавляет Вас от необходимости уменьшать значение регистра CX. Общая форма этих операторов

```
$ENDDO [условие[,AND/OR]][,LONG]
$ENDDO [ LOOPx][,LONG]
$LEAVE условие[,AND/OR][,LONG]
```

ПАРАМЕТР LOOP

Для циклов с заданным числом повторений, счетчиком которых служит регистр CX, в программе SALUT можно за счет применения формы $ENDDO LOOP опустить команду DEC CX. Иначе говоря, вместо формы

```
$DO
    ...
    ...
    DEC CX
$ENDDO CXZ
```

Вы можете использовать более простую форму

```
$DO
    ...
    ...
$ENDDO LOOP
```

Она заставляет программу SALUT заменить оператор $DO на метку, а оператор $ENDDO – на команду LOOP метка. Как указано в разд. 3.7, команда LOOP уменьшает значение регистра CX на 1 и заставляет микропроцессор 8088 передать управление метке, если значение CX отлично от нуля.

Чтобы учитывать при принятии решения об окончании цикла состояние флага нуля ZF, пользуйтесь одним из следующих операторов:

Оператор $ENDDO – уменьшает значение регистра CX и возвращает управление на начало цикла, если значение регистра CX не равно 0 и флаг ZF равен 1. Это позволяет обнаруживать первый ненулевой результат в серии операций.

Оператор $ENDDO LOOPNE уменьшает значение регистра CX и возвращает управление на начало цикла, если значение регистра CX не равно 0 и флаг ZF равен нулю. Это позволяет обнаруживать первый нулевой результат в серии операций.

Оператор $ENDDO LOOPZ является альтернативной формой оператора $ENDDO LOOPE.

Оператор $ENDDO LOOPNZ является альтернативной формой оператора $ENDDO LOOPNE.

ОПЕРАЦИИ AND И OR

Операция AND завершает цикл, если несколько условий выполняются одновременно, а операция OR завершает цикл, если выполняется *любое* из нескольких условий. Например, ниже показана общая форма структуры DO, которая завершает цикл, если в ответ на приглащение к вводу пользователь ответит "Нет", набрав на клавиатуре "Н" или "н":

```
$DO
   ...
   ...
   (Выдать на экран "Продолжить? (Д/Н)")
   (Считать ответ пользователя в переменную ANSWER)
   CMP ANSWER,"Н"
   $ENDDO E,OR
      CMP ANSWER,"н"
   $ENDDO E
```

11.4. СТРУКТУРА SEARCH

Структура SEARCH (искать) похожа на структуру DO, но предусматривает два различных варианта завершения цикла: *успешный* и *безуспешный*. Это удобно для тех приложений, где требуется знать, *какое* из условий привело к завершению цикла. Например, Вам может быть необходимо узнать, завершилась операция ввода-вывода успешно или была прекращена из-за возникновения ошибки.

Как и структура DO, структура SEARCH имеет три формы:

Структура SEARCH UNTIL (искать ... до) повторяет блок команд до тех пор, пока одно из двух условий в конце этого блока не станет истинным.

Структура SEARCH WHILE (искать, пока не) повторяет блок команд, пока оба условия ложны. Если какое-либо из них изначально истинно, то блок команд не исполняется ни разу.

Структура SEARCH COMPLEX (искать комплексно) представляет собой комбинацию структур SEARH UNTIL и SEARCH WHILE, включающую в себя блок команд до проверок условий и блок команд после них. Она выполняет первый блок, затем проверяет условия. Если хотя бы одно из них истинно, то цикл завершается. В противном случае исполняется второй блок команд и процесс повторяется заново.

Примечание. Как и в случае структур DO, слова UNTIL, WHILE и COMPLEX использованы только в описательных целях; *не набирайте* их в текстах своих программ.

СТРУКТУРА SEARCH UNTIL

Эта структура имеет общий вид

```
$SEARCH
    A        (Основной блок)
$EXITIF     условие1
    B        (Если условие1 истинно, то выполнить и выйти из цикла)
$ORELSE
    C        (Если условие1 ложно, то выполнить)
$ENDLOOP    условие2
    D        (Если условие2 истинно, то выполнить и выйти из цикла)
$ENDSRCH
```

где **условие1** и **условие2** – условия завершения цикла. Здесь **условие1** соответствует успешному завершению (Вы нашли то, что искали), а **условие2** – безуспешному. На рис. 11.3, *а* показана блок-схема структуры SEARCH UNTIL.

Учтите, что основной блок А всегда выполняется по крайней мере один раз. Блоки В и D выполняются соответственно при успешном и безуспешном завершении операций (Вы можете пользоваться ими для выдачи сообщений), а блок С содержит команды, устанавливающие значения флагов для проверки оператором $ENDLOOP.

Например, в экзаменационной программе можно воспользоваться структурой SEARCH UNTIL и дать студенту три попытки выбрать из предложенных ему вариантов ответа правильный. Ее применение может иметь следующий общий вид:

```
      MOVE CX,3       ;Установить счетчик
          (Изобразить на экране вопрос)
$SEARCH
          (Прочитать ответ студента)
          (Установить ZF = 1, если ответ правилен)
$EXITIF Z
          (Изобразить на экране "Поздравляем, Вы правы!")
$ORELSE
          (Изобразить на экране "Простите, Вы ошиблись. Попробуйте
          снова")
$ENDLOOP LOOP
         (Изобразить на экране "Все три ответа неправильные")
         (Изобразить на экране "Правильный ответ:" и указать ответ)
$ENDSRCH
```

СТРУКТУРА SEARCH WHILE

Эта структура имеет общий вид

```
$SEARCH
$LEAVE     условие1
   A       (Если условие1 ложно, то выполнить)
$EXITIF    условие2
   B       (Если условие2 истинно, то выполнить и выйти из цикла)
$ORELSE
   C       (Если условие2 ложно, то выполнить)
$ENDLOOP
   D       (Если условие1 истинно, то выполнить и выйти из цикла)
$ENDSRCH
```

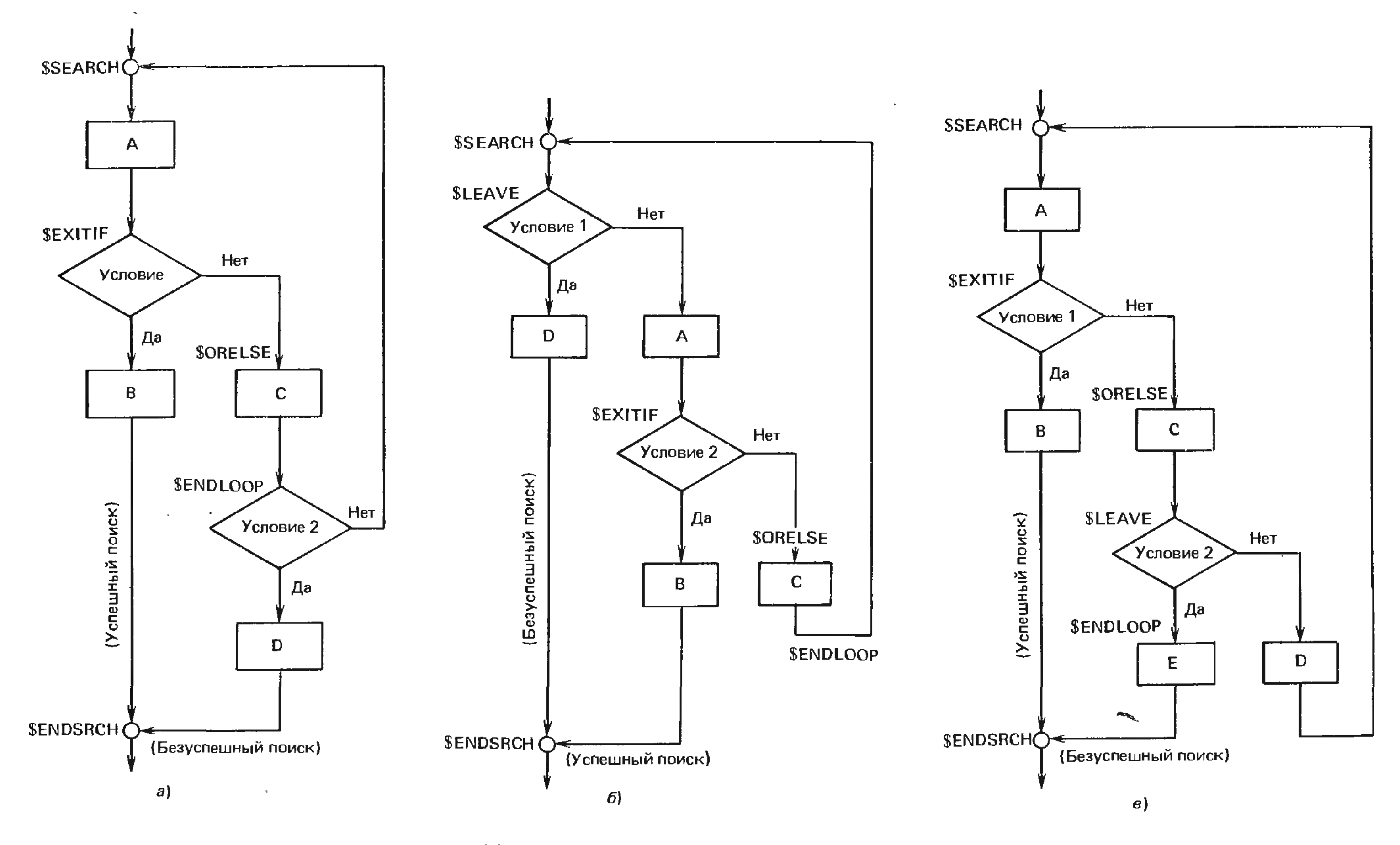


Рис. 11.3. Блок-схемы структур SEARCH UNTIL (*а*), SEARCH WHILE (*б*), SEARCH COMPLEX (*в*)

На рис. 11.3, *б* показана блок-схема структуры SEARCH WHILE.

Так как структура SEARCH WHILE осуществляет проверку до выполнения других операторов, то ее можно использовать для обхода всей структуры, если начальные условия не выполнены. Например, следующий фрагмент осуществляет поиск в таблице первого нулевого значения и обходит поиск, если таблица пуста:

```
      MOV CX,TABLE_LENGTH   ;Установить счетчик
      MOV BX,0              ;Установить указатель на первый элемент
$SEARCH
$LEAVE CXZ
      CMP TABLE[BX],0       ;Сравнить с нулем
$EXITIF E
      (Изобразить на экране "Найден нуль")
$ORELSE
      INC BX
$ENDLOOP LOOP
      (Изобразить "Нулей нет")
$ENDSRCH
```

СТРУКТУРА SEARCH COMPLEX

Эта структура имеет общий вид

```
$SEARCH
    A
$EXITIF  условие1
    B    (Если условие1 истинно, то выполнить и выйти из цикла)
$ORELSE
    C    (Если условие1 ложно, то выполнить)
$LEAVE   условие2
    D    (Если условие2 ложно, то выполнить)
$ENDLOOP
    E    (Если условие2 истинно, то выполнить и выйти из цикла)
$ENDSRCH
```

На рис. 11.3, *в* показана блок-схема структуры SEARCH COMPLEX. Обратите внимание на то, что структура SEARCH COMPLEX аналогична структуре SEARCH UNTIL, но выполняет дополнительный блок D, если **условие 2** ложно.

ДОПОЛНИТЕЛЬНЫЕ ОПЕРАНДЫ

Некоторые операторы структуры SEARCH могут иметь дополнительные операнды типа тех, что описывались для структур IF и DO. Они имеют следующие общие формы:

```
$EXITIF условие[,AND/OR][,LONG][,NUL]
$ORELSE [LONG]
$ENDLOOP [условие[,AND/OR]][,LONG]
$ENDLOOP [LOOPx],[LONG]
$LEAVE условие[,AND/OR][,LONG]
```

Единственный новый операнд – NUL, который можно использовать в сочетании с оператором $EXITIF в ситуации, когда в структуре SEARCH надо опустить блок B (*успешный поиск*). Конечно, если блока B нет, то можно опустить и оператор $ORELSE. Структура функционирует и в том случае, если блок B пуст, но указание операнда NUL ускоряет ее выполнение.

Таблица 11.2. **Допустимое использование условий NCXZ и CXZ**

Структурный оператор	NCXZ	CXZ
$IF условие	Да	Нет
$IF условие, OR	Нет	Да
$IF условие, AND	Да	Нет
$LEAVE **условие**	Нет	Да
$LEAVE **условие**, OR	Нет	Да
$LEAVE **условие**, AND	Да	Нет
$EXITIF **условие**	Да	Нет
$EXITIF **условие**, AND	Да	Нет
$EXITIF **условие**, OR	Нет	Да
$EXITIF **условие**, AND, NUL	Да	Нет
$EXITIF **условие**, OR, NUL	Нет	Да
$EXITIF **условие**, NUL	Нет	Да
$ENDLOOP **условие**	Да	Нет
$ENDLOOP **условие**, AND	Да	Нет
$ENDLOOP **условие**, OR	Нет	Да
$ENDDO **условие**	Да	Нет
$ENDDO **условие**, AND	Да	Нет
$ENDDO **условие**, OR	Нет	Да

11.5. ОГРАНИЧЕНИЯ НА ИСПОЛЬЗОВАНИЕ УСЛОВИЙ NCXZ И CXZ

Как упоминалось выше, программа SALUT транслирует структурные операторы в их эквиваленты на языке ассемблера. В частности, она преобразует условные операторы в команды Jx или JMP, а операторы $ENDIF, $DO, $SEARCH и $ENDSRCH – в метки. При этом программа SALUT во многих случаях должна преобразовывать условный оператор в противоположную команду условной передачи управления. Например, она преобразует оператор $IF A в команду JNA L1 (где L1 – метка, стоящая на месте оператора $ENDIF). Но из-за этого при применении условия CXZ могут возникнуть проблемы, поскольку команды JNCXZ нет. В табл. 11.2 перечислены те операторы, в которых могут использоваться условия CXZ и NCXZ, и показано, в каких случаях эти условия допустимы.

11.6. СОСТАВЛЕНИЕ СТРУКТУРИРОВАННЫХ ПРОГРАММ

ПРОЦЕДУРА

При составлении структурированных программ можно пользоваться теми же приемами, что и при составлении обычных программ, с той лишь разницей, что текст программы надо обработать программой SALUT для преобразования структурных операторов в стандартные команды на ассемблера. Кроме того,

программа SALUT будет формировать Вашу исходную программу, располагая элементы операторов по определенным столбцам. Таким образом, надо выполнить следующие шаги:

1. Набрать текст программы с помощью программы обработки текстов, редактора или программы EDLIN. Дать ему имя вида **имя_файла**.SAL, где SAL – аббревиатура от Structured Assembly Language (структурированный язык ассемблера).

Важное указание: при вводе текста программы набирайте все самостоятельные комментарии и метки, начиная со столбца 1, а структурные операторы, непомеченные команды и непомеченные псевдооператоры – с некоторым отступом. Например, Вы можете набирать их со столбца 2, нажимая на клавишу пробела или (при работе с программой EDLIN) со столбца 9, нажимая на клавишу TAB.

2. Обработайте файл типа SAL программой SALUT. В результате Вы получите файл с форматированной исходной программой **имя_файла**.SAL и промежуточный рабочий файл **имя_файла**.ASM. Программа SALUT переименует созданный Вами файл с неформатированной программой в **имя_файла**.BAK.

3. Оттранслируйте промежуточный рабочий файл с помощью Ассемблера MASM.

4. Удалите промежуточный рабочий файл типа ASM. При желании можно удалить и файл типа BAK.

5. Воспользуйтесь загрузчиком для создания исполняемого модуля.

РАЗРАБОТКА СТРУКТУРИРОВАННЫХ ПРОГРАММ ПО МЕТОДУ "СВЕРХУ ВНИЗ"

В гл. 2 мы обсуждали метод разработки "сверху вниз", согласно которому программа формируется последовательной вставкой деталей в ее первоначальный набросок (состоящий чаще всего лишь из комментариев). Этот метод можно применить и для разработки структурированных программ. Для этого сначала набирайте управляющие операторы логических структур, а затем вставляйте команды, которые должны быть между ними. Чтобы построить, например, структуру IF, надо начать с операторов

```
$IF         ;Если указано правильное значение
$ELSE       ;Так как указано неправильное значение
$ENDIF      ;Конец проверки значения на правильность
```

Эта конструкция пока еще не может быть оттранслирована должным образом, поскольку в операторе $IF надо указывать условие. Добавляя условие, получаем

```
$IF E       ;Если указано правильное значение
$ELSE       ;Так как указано неправильное значение
$ENDIF      ;Конец проверки значения на правильность
```

Теперь мы имеем условие "равно", которому должно предшествовать сравнение значений. Вставляя это сравнение, получаем

```
CMP AX,100      ;Проверить на совпадение с требуемым значением
$IF E           ;Если указано правильное значение
$ELSE           ;Так как указано неправильное значение
$ENDIF          ;Конец проверки значения на правильность
```

Обратите внимание на то, что теперь мы получили нечто приемлемое как для трансляции, так и для исполнения. Поэтому мы на какое-то время можем оставить

этот фрагмент программы, а позже вернуться к нему для того, чтобы заполнить "истинную" и "ложную" части проверки. В конце концов эскиз, который мы получили, вполне пригоден для документирования назначения этого фрагмента, а детали можно добавить и позже.

Важным моментом этого примера является то обстоятельство, что *комментарии* появились в тексте одновременно с командами. Лучше всего документировать программу в момент ее написания, поскольку именно в это время Вы лучше всего понимаете ее. Никогда не говорите себе "вставлю комментарии позже", так как, вероятнее всего, Вы этого не сделаете. Но если "позже" и наступит, то, может быть, Вы и не вспомните, что требовалось сделать.

ИСПОЛЬЗОВАНИЕ ПРОГРАММЫ SALUT

Для вызова программы SALUT получите на экране приглашение к вводу B>, а затем введите команду вида

```
B>a:salut имя_файла
```

(Если Вы не указали расширение имени файла, то программа SALUT добавит к **имя_файла** расширение SAL.)

Когда программа SALUT завершит свою работу, на установленном в дисководе B диске окажется три файла: первоначальная исходная программа **имя_файла**.BAK, переформатированная версия исходной программы **имя_файла**.SAL и промежуточный рабочий файл **имя_файла**.ASM. Теперь можно обычным образом оттранслировать файл **имя_файла**.ASM, а затем загрузить объектный модуль **имя_файла**.OBJ и получить исполняемый модуль **имя_файла**.EXE.

ПАКЕТ КОМАНД ДЛЯ ПРОГРАММЫ SALUT

Вообще говоря, после завершения работы программы SALUT Вам уже не нужен файл с первоначальной исходной программой (получившей расширение BAK). А после завершения трансляции Вам не нужен и файл с расширением ASM (если в программе имелись ошибки, то Вам надо исправлять не файл с расширением ASM, а файл с расширением SAL.) Поэтому Вам потребуется удалить эти два файла. Чтобы не делать это каждый раз вручную, создайте *пакет команд*, который вызывает программу SALUT и Макроассемблер, а затем удаляет файлы с расширением BAK и ASM. Для этого (предполагая, что Вы хотите дать пакету команд имя ASMSAL.BAT) наберите указанные ниже строки, завершая каждую из них нажатием на клавишу возврата каретки:

```
B>copy con: a:asmsal.bat
  a:salut %1
  erase %1.bak
  a:masm %1,,;
  erase %1.asm
  (нажмите клавишу F6)
```

Для исполнения этого пакета введите команду вида

```
B> a:asmsal имя_файла
```

Учтите, что третья команда пакета, к сожалению, автоматически создает файл с листингом исходной программы (с расширением LST). Если он не нужен, создайте другой пакет команд, указав команду a:masm % 1; вместо этой строки.

ПЕРЕФОРМАТИРОВАНИЕ ИСХОДНЫХ ТЕКСТОВ ПРОГРАММОЙ SALUT

Если Вы не зададите иные параметры, то программа SALUT будет переформатировать исходную программу следующим образом:

1. Начальной позицией *меток* и *самостоятельных комментариев* будет столбец 1.
2. Начальной позицией *структурных операторов* и *мнемокодов* команд будет столбец 9.
3. Начальной позицией *операндов* будет столбец 17.
4. Начальной позицией *комментариев* будет столбец 41.
5. *Внутри структур* операторы смещаются на четыре позиции вправо.

При желании можно задать другие начальные позиции. Детали см. в руководстве по Макроассемблеру.

Чтобы посмотреть, как выполняется это переформатирование, вставьте в программу следующий фрагмент:

```
MOV CX,4              ;Выполнить четыре операции
MOV BX,0              ;Указать на первый элемент
$DO
MOV DEST[BX],0        ;Обнулить текущий байт
INC BX                ;Перейти к следующему байту
$ENDDO LOOP
```

После того как программа SALUT переформатирует его, фрагмент примет следующий вид:

```
MOV     CX,4                            ;Выполнить четыре операции
MOV     BX,0                            ;Указать на первый элемент
$DO
    MOV     DEST[BX],0                  ;Обнулить текущий байт
    INC     BX                          ;Перейти к следующему байту
$ENDDO  LOOP
```

ПЕРЕФОРМАТИРОВАНИЕ НЕСТРУКТУРИРОВАННЫХ ПРОГРАММ

Учтите, что у программы SALUT функция переформатирования отделена от функции обработки структур. Следовательно, если Вы и не пользуетесь структурами (стыдитесь!), то тем не менее можете вызвать программу SALUT для переформатирования своей программы, написанной на языке ассемблера. Для этого наберите текст программы как обычно (скажем, разделяя элементы операторов одним пробелом или символом табуляции), затем сохраните ее в файле и введите команду вида

```
B>a:salut имя_файла.asm,имя_файла.asm,nul
```

Программа SALUT переформатирует файл **имя_файла**.ASM и переименует Вашу неформатированную версию в **имя_файла**.BAK.

ГЛАВА 12. МАТЕМАТИЧЕСКИЙ СОПРОЦЕССОР 8087

Математический сопроцессор 8087 фирмы Intel представляет собой микросхему, выполняющую сложные математические вычисления. Он разработан как средство расширения арифметических возможностей микропроцессоров 8088 и 8086. В связи с этим в персональной ЭВМ фирмы IBM (а также в совместимых с ней моделях) предусмотрена дополнительная панель для установки сопроцессора 8087.

Как Вам известно, микропроцессор 8088 сам может выполнять арифметические операции, но довольно ограниченные. Он может оперировать только пятизначными (двухбайтовыми) целыми числами и обеспечивает выполнение всего четырех основных операций: сложения, вычитания, умножения и деления. А сопроцессор 8087 способен выполнять широкий набор арифметических, логарифмических и тригонометрических операций над целыми и вещественными числами, имеющими до 18 десятичных разрядов. И поскольку команды сопроцессора 8087 реализованы аппаратно, то он способен кардинально ускорить выполнение этих операций по сравнению с микропроцессором 8088. Например, у программы для микропроцессора 8088 64-битовое умножение двоичных чисел может занять 210 мкс, в то время как сопроцессору 8087 на эту операцию потребуется только 30 мкс!

Микросхема 8087 называется *сопроцессором* потому, что при исполнении программ она работает в сочетании с основным микропроцессором 8088. Когда исполняется программа, рассчитанная на использование сопроцессора 8087, то микропроцессор 8088 исполняет те команды, которые он распознает как свои, а сопроцессор 8087 – те, которые он распознает как свои. Каждый занимается только собственными командами и игнорирует остальные. (Представьте себе прораба, у которого два помощника – итальянец и испанец. Итальянец выполняет только те приказы, которые прораб отдает по-итальянски, а испанец – только те, которые отдаются по-испански.)

Программы, написанные на языке высокого уровня наподобие Бейсика или Паскаля, используют сопроцессор 8087 автоматически. Однако язык ассемблера требует, чтобы Вы или вставляли команды сопроцессора 8087 в качестве операнда команды микропроцессора 8088 ESC (escape), или же купили Ассемблер, который способен распознать команды сопроцессора 8087 (например, программу Макроассемблер из комплекта Макроассемблера версии 2 фирмы IBM).

В этой главе дается краткий обзор свойств сопроцессора 8087. По поводу деталей обратитесь к руководству по Макроассемблеру или закажите копию прекрасной книги Ричарда Старца "8087 Application and Programming" (издательство Brady Communication Company, Inc., 1983). В качестве подспорья можно воспользоваться файлом FLINS.ASM, который находится на диске с Макроассемблером версии 2 и содержит примеры использования большинства команд сопроцессора 8087.

12.1. ВНУТРЕННИЕ РЕГИСТРЫ

Сопроцессор 8087 имеет восемь 80-битовых регистров данных, а также слово состояния и слово управления, каждое по 16 битов. Слово состояние похоже на регистр флагов микропроцессора 8088. Слово управления задает сопроцессору 8087 режимы обработки округления, превращения в бесконечность, а также точность вычислений.

Работа с регистрами данных сопроцессора 8087 существенно отличается от работы с регистрами микропроцессора 8088 тем, что обычно Вы не адресуетесь к ним индивидуально, а используете их как *стек*.

СТЕК СОПРОЦЕССОРА 8087

Напомним, что микропроцессор 8088 использует стек для хранения адресов возврата во время вызовов процедур и обработки прерываний. Кроме того, мы могли пользоваться стеком для сохранения значений регистров. Стек действует подобно стопке тарелок в кафетерии: последний элемент, который Вы поместили в него, будет первым, который Вы сможете из него извлечь. Регистры данных сопроцессора 8087 работают точно так же. А именно, Вы всегда помещаете числа в "верхний" регистр, и при этом содержимое стека сдвигается "вниз" на одну позицию. (Как и микропроцессор 8087, сопроцессор 8087 не перемещает содержимое стека, а просто изменяет значение указателя стека.)

Поскольку регистры данных функционируют как стек, то большинство команд сопроцессора 8087 неявным образом используют содержимое стека. Например, если Вы даете сопроцессору 8087 команду сложения, то он складывает значения двух чисел, находящихся в стеке, и помещает результат в стек. Если Вам приходилось работать с языком, ориентированным на стек (например, с языком FORTH), то Вы будете достаточно уверенно составлять программы для сопроцессора 8087; в противном случае Вам придется какое-то время привыкать к этой концепции.

ФОРМАТ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Регистры данных содержат числа в формате с *плавающей точкой*, который является научной версией обозначения чисел для ЭВМ. В ее обозначениях число −150 надо записывать как $-1.5*10^2$.

Регистры данных сопроцессора 8087 представляют такие значения, разделяя их на три поля: однобитовый знак числа, 15-битовую характеристику и 64-битовое значение (или *мантиссу*). Следовательно, при загрузке числа −150 в регистр данных в поле знака попадет 1, в поле характеристики – представление числа 2, а в поле мантиссы – 15.

К счастью, обычно нам не надо помнить, в каком виде сопроцессор 8087 хранит числа. Но нам надо знать о том, какими видами данных он может оперировать.

12.2. ТИПЫ ДАННЫХ

Сопроцессор 8087 может оперировать семью типами данных: тремя типами целых чисел (слово, короткое и длинное целое), тремя типами вещественных чисел (короткое, длинное и рабочее) и упакованными двоично-десятичными числами (табл. 12.1).

Среди целых типов двухбайтовое *целое слово,* соответствующее целому типу данных в языке Бейсик, полезно применять для индексации массивов и других структур данных. Так как целое слово может содержать значение от −32 768 до 32 767, то Вам, скорее всего, не часто придется иметь дело с коротким и длинным целыми типами.

Короткое вещественное значение и *длинное вещественное* значение соответствуют числовым значениям одинарной и двойной точности в языке Бейсик.

Таблица 12.1 **Типы данных сопроцессора 8087**

Тип данных	Число байтов	Число значащих цифр	Диапазон
Слово	16	4 или 5	−32768...32767
Короткий целый	32	9	$-2*10^{9}...2*10^{9}$
Длинный целый	64	18	$-9*10^{18}...9*10^{18}$
Короткий вещественный	32	6 или 7	$10^{-37}...10^{38}$
Длинный вещественный	64	15 или 16	$10^{-307}...10^{308}$
Рабочий вещественный	80	19	$10^{-4932}...10^{4932}$
Упакованный двоично-десятичный	80	18	18 десятичных цифр и знак

Короткие вещественные числа имеют около семи значащих цифр. Это означает, что короткие вещественные числа, отличающиеся только восьмым знаком, не различаются сопроцессором 8087. (Например, он считает числа 1,23456789 и 1,2345681 одинаковыми.) Таким образом, короткие вещественные значения удобны для хранения входных данных. А при выполнении вычислений следует пользоваться длинными вещественными значениями, что позволяет минимизировать накопление ошибок округления. Длинные вещественные значения имеют около 16 значащих цифр.

Рабочие вещественные значения представляют собой форму, в которой сопроцессор 8087 хранит числа в своих регистрах данных. Так как в этой форме на мантиссу отводится 64 бита, то все остальные типы данных могут быть преобразованы в эту форму без какой-либо потери точности. Рабочие вещественные значения занимают 80 битов и защищают пользователя от накапливания ошибок округления, а также от выхода за верхнюю или нижнюю границу диапазона допустимых вещественных чисел при промежуточных вычислениях.

Наконец, *упакованные двоично-десятичные* значения используются при экономических расчетах и обработке данных (в информационных системах). Они могут содержать до 18 значащих цифр; при этом цифры упаковываются по две в одном байте памяти. Вы должны помнить этот вид представления чисел по нашему обсуждению двоично-десятичных (BCD) значений в гл. 3.

12.3. СИСТЕМА КОМАНД

В системе команд сопроцессора 8087 можно выделить шесть групп: команды передачи данных, арифметические команды, команды сравнения, трансцендентные команды, команды загрузки констант и команды управления. В табл. 12.2 приведен общий вид их форматов (как обычно, квадратными скобками отмечены необязательные элементы команды) и дано их краткое описание.

Команды *передачи данных* загружают числа на вершину стека регистров данных сопроцессора 8087 и снимают их с вершины стека.

Арифметические команды обеспечивают выполнение четырех основных действий (сложение, вычитание, умножение и деление), а также вычисление ряда

Таблица 12.2. **Система команд сопроцессора 8087**

Команда	Действие
Команды передачи данных	
FBLD **источник**	Поместить в стек упакованное двоично-десятичное значение
FBSTP **приемник**	Извлечь из стека упакованное двоично-десятичное значение
FILD **источник**	Поместить в стек целое значение
FIST **приемник**	Скопировать вершину стека как целое значение
FISTP **приемник**	Извлечь из стека целое значение
FLD **источник**	Поместить в стек вещественное значение
FST **приемник**	Скопировать вершину стека как вещественное значение
FSTP **приемник**	Извлечь из стека вещественное значение
FXCH [**приемник**]	Обменяться значениями регистров
Арифметические команды	
FADD [**приемник** [, **источник**]]	Сложить вещественные значения
FADDP **приемник, источник**	Сложить вещественные числа и сдвинуть содержимое стека вверх
FIADD **источник**	Сложить целые значения
FSUB [**приемник** [, **источник**]]	Вычесть вещественные значения
FSUBP **приемник, источник**	Вычесть вещественные значения и сдвинуть содержимое стека вверх
FISUB **источник**	Вычесть целые значения
FSUBR [**приемник** [, **источник**]]	Выполнить обращенное вычитание вещественных значений
FSUBRP **приемник, источник**	Выполнить обращенное вычитание вещественных значений и сдвинуть содержимое стека вверх
FISUBR **источник**	Выполнить обращенное вычитание целых значений
FMUL [**приемник** [, **источник**]]	Перемножить вещественные значения
FMULP **приемник, источник**	Перемножить вещественные значения и сдвинуть содержимое стека вверх
FIMUL **источник**	Перемножить целые значения
FDIV [**приемник** [, **источник**]]	Разделить вещественные значения
FDIVP **приемник, источник**	Разделить вещественные значения и сдвинуть содержимое стека вверх
FIDIV **источник**	Разделить целые значения
FDIVR [**приемник** [, **источник**]]	Выполнить обращенное деление вещественных значений
FDIVRP **приемник, источник**	Выполнить обращенное деление вещественных значений и сдвинуть содержимое стека вверх
FIDIVR **источник**	Выполнить обращенное деление целых значений
FSQRT	Извлечь квадратный корень
FSCALE	Масштабировать по степеням двойки
FPREM	Найти частный остаток от деления
FRNDINT	Округлить до целого значения
FXTRACT	Извлечь характеристику и мантиссу
FABS	Найти абсолютное значение
FCHS	Изменить знак

Команда	Действие
Команды сравнения	
FCOM [**источник**]	Сравнить вещественные значения
FCOMP [**источник**]	Сравнить вещественные значения и сдвинуть содержимое стека вверх
FCOMPP	Сравнить вещественные значения и дважды сдвинуть содержимое стека вверх
FICOM **источник**	Сравнить целые значения
FICOMP **источник**	Сравнить целые значения и сдвинуть содержимое стека вверх
FTST	Проверить, не содержит ли вершина стека нулевое значение
FXAM	Проанализировать содержимое вершины стека
Трансцендентные команды	
F2XMI	Возвести 2 в степень X и вычесть 1
FYL2X	Умножить Y на $\log_2 X$
FYL2XP1	Умножить Y на $\log_2(X+1)$
FPTAN	Вычислить тангенс
FPATAN	Вычислить арктангенс
Команды загрузки констант	
FLDZ	Поместить в стек 0.0
FLD1	Поместить в стек 1.0
FLDPI	Поместить в стек π
FLDL2T	Поместить в стек $\log_2 10$
FLDL2E	Поместить в стек $\log_2 e$
FLDLG2	Поместить в стек lg2
FLDLN2	Поместить в стек ln2
Команды управления	
FLDCW **источник**	Загрузить слово управления
FSTCW **приемник**	Считать слово управления
FSTSW **приемник**	Считать слово состояния
FSAVE **приемник**	Сохранить значения регистров в памяти
FRSTOR **источник**	Восстановить значения регистров
FLDENV **источник**	Восстановить операционную среду
FSTENV **приемник**	Сохранить операционную среду в памяти
FWAIT	Перейти в состояние ожидания (приостановить работу микропроцессора 8088)
FINIT	Инициировать (перевести в начальное состояние) сопроцессор 8087
FENI	Включить систему прерываний
FDISI	Отключить систему прерываний
FCLEX	Обнулить флаги исключений
FINCSTP	Продвинуть указатель стека вверх
FDECSTP	Продвинуть указатель стека вниз
FFREE **приемник**	Освободить (очистить) регистр
FNOP	Холостой ход

распространенных функций, например квадратного корня и абсолютного значения. Команды вычитания и деления имеют две формы. В стандартной форме Вы вычитаете приемник из источника или делите приемник на источник. В обращенной форме Вы вычитаете источник из приемника или делите источник на приемник. Обращенные формы позволяют Вам оставлять результаты в ячейках памяти, которые могут служить только операндом-источником.

Команды *сравнения* сопоставляют число, находящееся на вершине стека, с содержимым другого регистра стека или ячейки памяти. Эти команды определяют, является ли число меньшим нуля, равным нулю или больше нуля.

Трансцендентные команды вычисляют логарифмические и тригонометрические функции. У сопроцессора 8087 предусмотрены только команды вычисления тангенса и арктангенса, но из них нетрудно вывести все остальные тригонометрические функции.

Команды *загрузки констант* позволяют занести в стек любую из семи констант: 0, 1, π, а также четыре логарифма. Во всех случаях сопроцессор 8087 дает им полную точность рабочих вещественных значений (19 значащих цифр).

Команды *управления* позволяют сохранять информацию о состоянии сопроцессора 8087, изменять режим округления результатов, включать и отключать прерывания, а также выполнять много других "внутренних дел". Одна из команд, FWAIT, генерирует команду микропроцессора 8088 WAIT, которая предохраняет его от доступа к ячейке памяти, используемой сопроцессором 8087.

12.4. ПРОГРАММИРОВАНИЕ СОПРОЦЕССОРА 8087 НА МАКРОАССЕМБЛЕРЕ

Программы, расчитанные на сопроцессор 8087, можно разрабатывать так же, как и программы для микропроцессора 8088, только Вам надо сообщить Ассемблеру, что Вы пользуетесь командами сопроцессора 8087. Это можно сделать одним из следующих способов:

указать команду .8087 в самом начале основной программы – непосредственно за оператором TITLE;

дать команду трансляции программы в форме

```
masm имя файла/R
```

КОНСТАНТЫ

В разд. 2.3 мы описали четыре вида констант, которые допускаются Ассемблером при обычном программировании для микропроцессора 8088: двоичные, десятичные, шестнадцатеричные и символьные (литералы). При работе с сопроцессором 8087 можно использовать две дополнительные формы констант:

1. *Десятичная вещественная константа* – число с десятичной точкой; например 3.14159.

2. *Десятичная константа в научно-инженерном формате* – десятичное число, за которым следует E и значение степени; например 2.654E–12.

ПСЕВДООПЕРАТОРЫ ОПРЕДЕЛЕНИЯ ДАННЫХ

В разд. 12.2 мы описали семь типов данных, воспринимаемых сопроцессором 8087: три целых (слово, короткое и длинное целое), три вещественных (короткое, длинное и рабочее) и упакованный двоично-десятичный. Макроассем-

блер MACRO располагает следующими псевдооператорами определения данных, позволяющих присваивать начальные значения переменным каждого типа:

Для целых слов, занимающих 16 битов, используется псевдооператор DW (define word – определить слово).

Для коротких целых и коротких вещественных значений, занимающих 32 бита, используется псевдооператор DD (define doubleword – определить двойное слово).

Для длинных целых и длинных вещественных значений, занимающих 64 бита, используется псевдооператор DQ (define quadword – определить учетверенное слово).

Для рабочих вещественных значений и упакованных целых значений, занимающих 80 битов, используется псевдооператор DT (define tenbytes – определить десяток байтов).

Обратите внимание на новые псевдооператоры DQ и DT. Ниже приведены примеры каждой формы констант:

```
WORD_INTEGER   DW   16483
SHORT_INTEGER  DD   145897
LONG_INTEGER   DQ   42765844
SHORT_REAL     DD   3.14159
LONG_REAL      DQ   1.0E-14
TEMP_REAL      DT   1.5E-16
PACKED_DEC     DT   -1457594442553
```

12.5. ЗАКЛЮЧЕНИЕ

Математический сопроцессор 8087 представляет собой мощное устройство, которое может кардинально улучшить возможности Вашей ЭВМ в части выполнения математических вычислений. Он воспринимает семь различных типов данных: три целых, три вещественных и упакованный двоично-десятичный, что дает максимальную гибкость при проведении любых требуемых вычислений. Каким бы типом данных Вы не пользовались, сопроцессор 8087 выполняет все внутренние действия над значениями в 80-битовом формате с плавающей точкой, что обеспечивает точность до 19 значащих цифр. Это не только приводит к высокой точности результатов, но и делает маловероятным выход за верхнюю и нижнюю границы диапазона допустимых значений.

У сопроцессора 8087 имеются команды для сложения, вычитания, умножения и деления как целых, так и вещественных чисел. Он имеет набор команд для вычисления логарифмических и тригонометрических функций, а также специальные команды для выполнения часто встречающихся вычислений, например для извлечения квадратного корня или определения абсолютного значения. Самая медленная команда исполняется за 190 мкс, а самая быстрая – за 1 мкс, но большинство команд исполняется за время от 18 до 40 мкс.

Поскольку сопроцессор 8087 имеет стек-ориентированную архитектуру, то программирование для него на языке ассемблера может потребовать выработки определенных навыков. Однако то небольшое количество времени, которое Вам придется на это затратить, с лихвой компенсирует муки написания процедур для выполнения арифметических операций над числами повышенной точности.

ОТВЕТЫ К УПРАЖНЕНИЯМ

ВВЕДЕНИЕ

1. а) 1100 б) 10001 в) 101101 г) 100100
2. а) 8 б) 21 в) 31
3. а) 8 б) 15 с) 1F
4. Шестнадцатеричное значение D8 может представлять число без знака 216 или со знаком –40. Чтобы получить число со знаком, надо преобразовать шестнадцатеричное число в двоичное (11011000), затем обратить каждый бит (получая 00100111) и добавить 1 (получая 00101000 или десятичное значение 40).

ГЛАВА 1

1. Системы команд микропроцессоров 8088 и 8086 идентичны.
2. При вычислении физического адреса микропроцессор 8088 автоматически добавляет четыре нуля к номеру блока, получая адрес блока. Следовательно, вместо значения 4000H он использует значение 40000H и вычислит физический адрес как

(физический адрес) = 2H + 40000H = 40002H

3. Если регистр AX содержит 1A2BH, то регистр AL (его младший байт) содержит 2BH.
4. Переменные обычно хранятся в сегменте данных, поэтому для доступа к ним используется регистр сегмента данных DS.
5. Бит 7, флаг знака SF, устанавливается равным 1, если вычитание приводит к отрицательному результату.

ГЛАВА 2

1. Ассемблер зарезервирует 1 байт для переменной VAR1, 10 байтов (5 слов=10 байтов) для переменной VAR2 и 10 байтов для переменной VAR3, всего – 21 байт.
2. Ассемблер не поместит никакого значения в переменную VAR1. Операнд "?" сообщает ему, что надо всего лишь *зарезервировать* для нее один байт. Ваша программа сама должна поместить значение в эту переменную.

Учтите, что в данном случае Ассемблер поместит значение только в одну ячейку, а именно, поместит 20 в пятое слово переменной VAR2. Значения всех остальных ячеек останутся "неопределенными".

3. Оператор "=" допускает последующее переопределение константы, в то время как EQU – нет.
4. Переменная CONST – *байтовая* и не может содержать значение, превышающее 255.
5. Каждая процедура должна начинаться с оператора PROC и заканчиваться оператором ENDP.
6. Процедуру с атрибутом NEAR можно вызывать только из сегмента, в котором она определена. А процедуру с атрибутом FAR можно вызвать из любого сегмента программы.
7. Первая процедура должна иметь атрибут FAR, поскольку вызывающая ее программа (операционная система DOS или отладчик DEBUG) находится в другом сегменте.

8. Оператор ASSUME указывает Ассемблеру, что для адресации каждой метки из сегмента CSEG надо пользоваться регистром CS. Другими словами, он идентифицирует CSEG как сегмент команд, а не сегмент данных, сегмент стека или дополнительный сегмент.

9. Загрузчик создает перемещаемый исполняемый модуль, который операционная система DOS может поместить в любую подходящую часть памяти. Это освобождает Вас от необходимости решать, в какое место памяти надо поместить программу. Кроме того, загрузчик объединяет два или более объектных модуля.

ГЛАВА 3

1. Этой командой является команда MOV ES:SAVE_AX, AX.

2. Эта последовательность команд заносит 0 в первую ячейку сегмента данных (адресуемую с помощью регистра BX) и в первую ячейку сегмента стека (адресуемую с помощью регистра BP).

3. а. Фрагмент ошибочен: константа не может быть операндом-приемником.

б. Фрагмент корректен, но поскольку начальное значение переменной TEMP не присвоено, то в регистре AL Вы получите "мусор".

в. Фрагмент ошибочен: нельзя занести слово в байтовую переменную.

г. Фрагмент ошибочен: команду MOV нельзя использовать для прямого обмена данными между ячейками памяти.

д. Команда ошибочна: Ассемблер не воспринимает адресацию вида [BX] [BP]. Правильные форматы операндов см. в табл. 3.1.

4. Приведенные ниже команды обнуляют регистр AX:

```
SUB AX,AX
MOV AX,0
```

5. Действие этих команд одинаково. Обе загружают смещение адреса ячейки TABLE+4 в регистр BX. Однако команда LEA короче и нагляднее.

6. Следующий цикл вычитает V2 из V1:

```
        MOV CX,3              ;Счетчик слов = 3
        MOV BX,0              ;Смещение = 0
        CLC                   ;Обнулить флаг переноса CF
NEXT:   MOV DX,V2[BX]         ;Вычитать слова
        SBB V1[BX],DX
        INC BX                ; и адресоваться к следующему слову
        INC BX
        LOOP NEXT
```

Обратите внимание на то, что после каждого вычитания регистр BX увеличивается на 2, поскольку слова памяти отстоят друг от друга на два байта. Мы воспользовались двумя командами INC вместо одной команды ADD, чтобы не воздействовать на флаг переноса CF, который используется командой SBB.

7. Эта команда MUL вызовет генерацию ошибки: нельзя умножать на непосредственное значение.

8. Результаты таковы:

а) (AX)=0220H;

б) (AX)=5335H;

в) (AX)=5115H;

г) (AX)=0EDCBH;

д) (AX)=1234H; (поскольку команда TEST воздействует только на флаги).

9. Нормализация содержимого регистра AX выполняется последовательностью команд

```
          TEST AX,0FFFFH
          JZ   NORM          ;Выйти, если (AX)=0
          MOV  CX,15         ;Установить счетчик для 15 сдвигов
NEXT BIT: IS   NORM          ;Выйти, если бит 15 равен 1
          SHL  AX,1          ;Иначе сдвинуть AX влево на один бит
          LOOP NEXT BIT
NORM:     ...
          ...
```

10. Если Вы решили, что этот фрагмент выполняет вычитание 30 из значения регистра AX, взгляните на него еще раз. Команда LOOP уменьшает значение регистра CX на единицу, а затем передает управление метке START, пока значение регистра CX не станет равным нулю. Но команда MOV каждый раз заново загружает 3 в регистр CX, поэтому его значение никогда не станет нулевым. Подобный вид бесконечного цикла является типичной ошибкой программирования. Будьте внимательны.

ГЛАВА 4

1. Приведенная ниже процедура извлекает квадратный корень вычитанием последовательных нечетных чисел:

```
SR32    PROC
        PUSH   AX        ;Сохранить в стеке исходное число
        PUSH   DX
        PUSH   CX        ; и значение регистра CX
        MOV    BX,1      ;Выполнить начальные установки: (BX) = 1
        SUB    CX,CX     ; и (CX) = 0
AGAIN:  SUB    AX,BX     ;Вычесть текущее нечетное число из AX
        SBB    DX,0      ; и DX
        JC     DONE      ;При вычитании возник заем?
        INC    CX        ; Нет. Увеличить квадратный корень на 1,
        ADD    BX,2      ;  вычислить следующее нечетное число
        JMP    AGAIN     ;  и перейти к следующему вычитанию
DONE:   MOV    BX,CX     ; Да. Передать результат через регистр BX
        POP    CX        ;  и восстановить значения регистров
        POP    DX
        POP    AX
        RET
SR32    ENDP
```

ГЛАВА 5

1. Эта модифицированная версия примера 5.1 может быть использована для образования списка "от нуля", а также для добавления нового элемента к уже существующему списку:

```
ADD_TO_UL   PROC
            PUSH   DI           ;Сохранить начальный адрес
            MOV    CX,ES:[DI]   ;Извлечь счетчик слов
            ADD    DI,2         ;Сделать DI указателем на первый элемент
            CMP    CX,0         ;Список пуст?
            JE     ADD_IT       ; Да. Значение будет первым элементом
            CLD                 ; Нет. Положить DF = 0
REPNE       SCASW               ;Значение уже находится в списке?
            JNE    ADD_IT
            POP    DI           ; Да. Восстановить начальный адрес
            RET                 ;  и выйти из процедуры
```

```
ADD_IT:      STOSW                   ; Нет. Добавить его в конец списка
             POP     DI              ;  и увеличить счетчик элементов
             INC     WORD PTR ES:[DI]
             RET
ADD_TO_UL    ENDP
```

2. Ниже приведена процедура поиска и замены:

```
; Эта процедура выполняет поиск значения размером в слово, содержаще-
; гося в регистре AX, в упорядоченном списке, находящемся в дополни-
; тельном сегменте. Если в списке найден элемент, совпадающий с этим
; значением, то его содержимое заменяется на значение регистра BX
; Значения регистров AX и BX не изменяются
;
REPLACE   PROC
          CALL   B_SEARCH        ;Искомое значение уже находится в списке?
          JC     QUIT            ; Нет. Выйти из процедуры
          MOV    ES:[SI],BX      ; Да. Заменить его на значение регистра BX
QUIT:     RET
REPLACE   ENDP
```

ГЛАВА 6

1. Следующая процедура измеряет промежуток времени между двумя нажатиями на клавиши. Она возвращает часы в регистре CH (на случай, если оператор отошел выпить кофе перед вторым нажатием), минуты в регистре CL, секунды в регистре DH и сотые доли секунды в регистре DL. Значения других регистров не изменяются.

```
TIME_KEYS   PROC
            PUSH   AX
            MOV    AH,7         ;При первом нажатии на клавишу
            INT    21H
            SUB    CX,CX        ; обнулить показания таймера
            SUB    DX,DX
            MOV    AH,2DH
            INT    21H
            MOV    AH,7         ;При втором нажатии на клавишу
            INT    21H
            MOV    AH,2CH       ; считать показания таймера
            INT    21H
            POP    AX           ;Восстановить значение регистра AX
            RET                 ; и выйти из процедуры
TIME_KEYS    ENDP
```

2. Автор добился минимального значения счетчика DX=10H, или 0,16 с. Однако за многие годы вино, женщины, песни и программирование на ЭВМ взяли свое, и Ваш результат. может оказаться лучше. В любом случае этот тест показывает, насколько машины быстрее людей. Процессор может выполнить *тысячи* команд за то время, пока Вы дважды нажимаете на клавишу!

3. Вероятно, наши сообщения заданы операторами

```
MSG1    DB    'Программа сортировки$'
MSG2    DB    'Нажмите любую клавишу для продолжения $'
```

Чтобы они выдались на отдельных строках, надо включить в текст символы возврата каретки и перехода на новую строку:

```
MSG1    DB    'Программа сортировки',0DH,0AH,'$'
MSG2    DB    'Нажмите любую клавишу для продолжения',0DH,0AH,'$'
```

4. Для изображения числа, находящегося в регистре CX, его надо преобразовать в ASCII-код, добавив к нему 30H. Требуемая программа состоит из сегмента данных, содержащего операторы

```
MSG_START DB 'Попробуте снова. У Вас осталось $'
MSG_END   DB  'истребителей.',0DH,0AH,'$'
```

и сегмента команд, содержащего операторы

```
LEA   DX,MSG_START      ;Изобразить начало сообщения
MOV   AH,9
INT   21H
MOV   DL,CL             ;Преобразовать число в ASCII-код
ADD   DL,30H
MOV   AH,6              ; и изобразить его
LEA   DX,MSG_END        ;Изобразить "хвост" сообщения
MOV   AH,9
INT   21H
```

ГЛАВА 7

1. Приведенная ниже процедура перемещает "птичку" на экране вдоль строки 20:

```
             EXTRN   DELAY:FAR
MOVE_BIRD    PROC
             PUSH    AX          ;Сохранить значения регистров
             PUSH    BX
             PUSH    CX
             PUSH    DX
             MOV     AH,15       ;Загрузить в ВХ номер активной страницы
             INT     10H         ; экрана
             MOV     AH,0        ;Выбрать текстовый ч/б режим 80*25
             INT     10H
             MOV     CX,1        ;Счетчик символов = 1
             MOV     DH,20       ;Начать в строке 20,
             MOV     DL,0        ; столбце 0
DSPLY_V:     MOV     BL,'v'      ;Изобразить на экране "v"
             CALL    NEXT_BIRD
             CMP     DL,80       ;Движение завершено?
             JE      BIRD_DONE
             MOV     BL,0C4H     ; Нет. Изобразить "тире"
             CALL    NEXT_BIRD
             CMP     DL,80       ;Движение завершено?
             JNE     DSPLY_V     ; Нет. Продвинуться дальше
BIRD_DONE:   POP     DX          ; Да. Восстановить значения регистров
             POP     CX
             POP     BX
             POP     AX
             RET                 ;  и выйти из процедуры
MOVE_BIRD    ENDP
;
; Эта процедура изображает "птичку" ("v" или "тире") в течение 0,1 с,
; затем стирает ее и продвигает указатель DL к следующему столбцу
;
NEXT_BIRD     PROC
             MOV     AH,2        ;Переместить курсор в следующую позицию
             INT     10H
             MOV     AH,10       ;Изобразить символ на экране
             MOV     AL,BL
             INT     10H
             CALL    DLY_TENTH   ;Выждать 0,1 с
             SUB     AL,AL       ; и стереть "птичку"
             MOV     AH,10
             INT     10H
             INC     DL          ;Указать на следующий столбец
             RET                 ; и возвратиться в вызвавшую процедуру
NEXT_BIRD     ENDP
;
; Эта процедура генерирует паузу в 0,1 с
;
```

```
DLY_TENTH     PROC
              SUB   AL,AL        ;Обнулить минуты
              SUB   BH,BH        ; и секунды
              MOV   BL,10        ;Положить сотые доли секунды = 10
              CALL  DELAY
              RET
DLY_TENTH     ENDP
```

Надо признаться, что формируемый этой программой образ не слишком похож на птицу, но это плата за использование псевдографики, а не точечной графики. Если Вам удастся сделать лучшую программу перемещения и изменения образа, то автор будет рад узнать об этом.

2. Приведенная ниже процедура, перемещающая случайным образом "улыбающуюся рожицу", представляет собой комбинацию процедур из примеров 6.1 и 7.2. "Рожица" изображается в течение 0,25 с между перемещениями.

```
              EXTRN  DELAY:FAR
RANG_FACE     PROC
              PUSH   AX           ;Сохранить значения регистров
              PUSH   BX
              PUSH   CX
              PUSH   DX
              MOV    AH,15        ;Загрузить в BX номер активной страницы
              INT    10H          ; экрана
              MOV    AH,0         ;Выбрать текстовый ч/б режим 80*25
              INT    10H
              MOV    CX,1         ;Счетчик символов = 1
INIT_LINE:    CALL   RAND_8       ;Выбрать случайный номер строки
              AND    DL,1FH
              CMP    DL,24        ; в пределах 0-24
              JA     INIT_LINE
              MOV    DH,DL        ; и загрузить его в регистр DH
              MOV    DL,0         ;Начать в столбце 0
NEXT_FACE:    MOV    AH,2         ;Переместить курсор в очередную позицию
              INT    10H
              MOV    AL,2         ;Изобразить улыбающуюся рожицу
              MOV    AH,10
              INT    10H
              CALL   DLY_QRTR     ;Выждать 1/4 с
              SUB    AL,AL        ; и стереть рожицу
              MOV    AH,10
              INT    10H
              MOV    AX,DX        ;Сохранить строку, столбец в регистре AX
LINE_IND:     CALL   RAND_8       ;Взять случайное число в пределах 0-2
              AND    DL,3
              CMP    DL,2
              JNE    CHK_3
              MOV    DX,AX        ;Оно равно 2. Ничего не делать
              JE     INC_COL
CHK_3:        JA     LINE_IND
              CMP    DL,1
              JNE    ITS_0
              MOV    DX,AX        ;Оно равно 1. Увеличить номер строки
              INC    DH
              CMP    DH,25        ; и выяснить, стал ли он < 25
              JE     EXIT_FACE
              JNE    INC_COL
ITS_0:        MOV    DX,AX        ;Оно равно 0. Уменьшить номер строки
              DEC    DH
              JS     EXIT_FACE    ; и выяснить, остался ли он положителен
INC_COL:      INC    DL           ;Указать на следующий столбец
              CMP    DL,80        ; и выйти из процедуры, если он = 80
              JNE    NEXT_FACE
```

```
EXIT_FACE:      POP     DX          ;Восстановить значения регистров
                POP     CX
                POP     BX
                POP     AX
                RET                 ; и выйти из процедуры
RAND_FACE       ENDP
;
; Эта процедура генерирует паузу в 1/4 с
;
DLY_QRTR        PROC
                SUB     AL,AL       ;Обнулить минуты
                SUB     BH,BH       ; и секунды
                MOV     BL,25       ;Положить сотые доли секунды = 25
                CALL    DELAY
                RET
DLY_QRTR        ENDP
;
; Эта процедура возвращает в регистре DX 16-битовое случайное число,
; но вызывающая процедура RAND_FACE использует только значение
; регистра DL
;
RAND_8          PROC
                PUSH    AX          ;Сохранить значения регистров AX и CX
                PUSH    CX
                MOV     AH,0        ;Считать показания таймера
                INT     1AH
                POP     CX          ; и выйти из процедуры
                POP     AX
                RET
RAND_8          ENDP
```

ПРИЛОЖЕНИЕ А. ПРЕОБРАЗОВАНИЕ ШЕСТНАДЦАТЕРИЧНЫХ ЧИСЕЛ В ДЕСЯТИЧНЫЕ И ОБРАТНО

Шестнадцатеричные позиции											
6		5		4		3		2		1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
7654		3210		7654		3210		7654		3210	
Байт				Байт				Байт			

HEX — шестнадцатеричная цифра; DEC — десятичное значение

Степени числа 2

2^n	n
256	8
512	9
1 024	10
2 048	11
4 096	12
8 192	13
16 384	14
32 768	15
65 536	16
131 072	17
262 144	18
524 288	19
1 048 576	20
2 097 152	21
4 194 304	22
8 388 608	23
16 777 216	24

$2^0 = 16^0$
$2^4 = 16^1$
$2^8 = 16^2$
$2^{12} = 16^3$
$2^{16} = 16^4$
$2^{20} = 16^5$
$2^{24} = 16^6$
$2^{28} = 16^7$
$2^{32} = 16^8$
$2^{36} = 16^9$
$2^{40} = 16^{10}$
$2^{44} = 16^{11}$
$2^{48} = 16^{12}$
$2^{52} = 16^{13}$
$2^{56} = 16^{14}$
$2^{60} = 16^{15}$

Степени числа 16

16^n	n
1	0
16	1
256	2
4 096	3
65 536	4
1 048 576	5
16 777 216	6
268 435 456	7
4 294 967 296	8
68 719 476 736	9
1 099 511 627 776	10
17 592 186 044 416	11
281 474 976 710 656	12
4 503 599 627 370 496	13
72 057 594 037 927 936	14
1 152 921 504 606 846 976	15

ПРИЛОЖЕНИЕ Б. НАБОР ASCII-СИМВОЛОВ ПЕРСОНАЛЬНОЙ ЭВМ IBM PC

Старшая часть (столбцы); Младшая часть (строки)

Десятичное значение →		0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
	Шестнадцатеричное значение →	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	Пустой символ (0)	►	Пробел	0	@	P	`	p	Ç	É	á	1/4	└	╨	∝	≡
1	1	☺	◄	!	1	A	Q	a	q	ü	Æ	í	1/2	┴	╤	β	±
2	2	☻	↕	"	2	B	R	b	r	é	FE	ó	3/4	┬	╥	γ	≥
3	3	♥	‼	#	3	C	S	c	s	â	ô	ú	│	├	╙	π	≤
4	4	♦	¶	$	4	D	T	d	t	ä	ö	ñ	┤	─	╘	Σ	⌠
5	5	♣	§	%	5	E	U	e	u	à	ò	Ñ	╡	┼	╒	σ	⌡
6	6	♠	▬	&	6	F	V	f	v	å	û	ª	╢	╞	╓	µ	÷
7	7	Гудок •	↨	'	7	G	W	g	w	ç	ù	º	╖	╟	╫	τ	≈
8	8	◘	↑	(	8	H	X	h	x	ê	ÿ	¿	╕	╚	╪	Φ	°
9	9	○	↓	)	9	I	Y	i	y	ë	Ö	⌐	╣	╔	┘	Θ	•
10	A	◙	→	*	:	J	Z	j	z	è	Ü	¬	║	╩	┌	Ω	·
11	B	♂	ESC ←	+	;	K	[	k	{	ï	¢	½	╗	╦	█	δ	√
12	C	♀	∟	,	<	L	\	l	¦	î	£	¼	╝	╠	▄	∞	η
13	D	♪	↔	—	=	M	]	m	}	ì	¥	¡	╜	═	▌	∅	²
14	E	♫	▲	.	>	N	^	n	~	Ä	Pts	«	╛	╬	▐	∈	■
15	F	☼	▼	/	?	O	_	o	△	Å	ƒ	»	┐	╧	▀	∩	Пустой символ (FF)

ПРИЛОЖЕНИЕ В. ВРЕМЯ ИСПОЛНЕНИЯ КОМАНД МИКРОПРОЦЕССОРОМ 8088

Вы можете пользоваться приведенными в этом приложении двумя таблицами для вычисления времени, которое потребуется микропроцессору 8088 для исполнения команд Вашей программы. В этих таблицах время исполнения команд дается в циклах тактового генератора. Чтобы преобразовать это время в наносекунды, умножьте его на 210.

В табл. В.2 указаны число *циклов*, которое требуется для исполнения каждой команды, а также число *байтов*, которое она занимает в памяти. Пользуясь табл. В.2, имейте в виду следующее:

1. Время исполнения некоторых команд удлиняется, если их операндами являются слова, а не байты. Для этих команд время исполнения указано в виде b(w), где b и w означают числа циклов для операндов-байтов и операндов-слов соответственно.

Таблица В.1. **Времена вычисления исполительного адреса**

Компоненты исполнительного адреса	Формат операнда	Число тактов[1]
Только сдвиг	сдвиг	6
	метка	
Только база или индекс	[BX]	5
	[BP]	
	[DI]	
	[SI]	
Сдвиг + база или индекс	[BX] + сдвиг	9
	[BP] + сдвиг	
	[DI] + сдвиг	
	[SI] + сдвиг	
База + индекс	[BX] [SI]	7
	[BX] [DI]	
	[BP] [SI]	8
	[BP] [DI]	
Сдвиг + база + индекс	[BX] [SI] + сдвиг	11
	[BX] [DI] + сдвиг	
	[BP] [SI] + сдвиг	12
	[BP] [DI] + сдвиг	

[1] При замене сегмента добавьте два такта.

Таблица В.2. **Времена исполнения команд**

Команда	Число тактов	Число байтов
AAA	4	1
AAD	60	2
AAM	83	1
AAS	4	1
ADC регистр,регистр	3	2
ADC регистр, память	9 (13)+EA	2–4
ADC память, регистр	16 (24)+EA	2–4
ADC регистр, непосредственный операнд	4	3–4
ADC память, непосредственный операнд	17 (25)+EA	3–6
ADC аккумулятор, непосредственный операнд	4	2–3
ADD регистр, регистр	3	2
ADD регистр, память	9 (13)+EA	2–4
ADD память, регистр	16 (24)+EA	2–4

Команда	Число тактов	Число байтов
ADD регистр, непосредственный операнд	4	3–4
ADD память, непосредственный операнд	17 (25)+EA	3–6
ADD аккумулятор, непосредственный операнд	4	2–3
AND регистр, регистр	3	2
AND регистр, память	9 (13)+EA	2–4
	16 (24)+EA	2–4
AND регистр, непосредственный операнд	4	3–4
AND память, непосредственный операнд	17 (15)+EA	3–6
AND аккумулятор, непосредственный операнд	4	2–3
CALL процедура-атрибут FAR	23	3
CALL процедура-атрибут NEAR	36	5
CALL указатель-память 16	29	2–4
CALL указатель-регистр 16	24	2
CALL указатель-память 32	57	2–4
CBW	2	1
CLC	2	1
CLD	2	1
CLI	2	1
CMC	2	1
CMP регистр, регистр	3	2
CMP регистр, память	9 (13)+EA	2–4
CMP память, регистр	9 (13)+EA	2–4
CMP регистр, непосредственный операнд	4	3–4
CMP память, непосредственный операнд	10(14)+EA	3–6
CMP аккумулятор, непосредственный операнд	4	2–3
CMPS строка_приемник, строка_источник	22 (30)	1
CMPS (повтор) строка_приемник, строка_источник	9+22 (30)/повтор	1
CWD	5	1
DAA	4	1
DAS	4	1
DEC регистр 16	2	1
DEC регистр 8	3	2
DEC память	15 (23)+EA	2–4
DIV регистр 8	80-90	2
DIV регистр 16	144-162	2
DIV память 8	(86-96)+EA	2–4
DIV память 16	(154-172)+EA	2–4
ESC непосредственный операнд, память	(8-35)+EA	2–4
ESC непосредственный операнд, регистр	2	2
HLT	2	1
IDIV регистр 8	101-112	2

Команда	Число тактов	Число байтов
IDIV регистр 16	165-184	2
IDIV память 8	(107-118)+EA	2—4
IDIV память 16	(175-194)+EA	2—4
IMUL регистр 8	80—98	2
IMUL регистр 16	128—154	2
IMUL память 8	(86—104)+EA	2—4
IMUL память 16	(138—164)+EA	2—4
IN аккумулятор, непосредственный-операнд 8	10 (14)	2
IN аккумулятор, DX	8 (12)	1
INC регистр 16	2	1
INC регистр 8	3	2
INC память	15 (23)+EA	2—4
INT 3	52	1
INT непосредственный-операнд 8 (не тип 3)	51	2
INTO	53 или 4	1
IRET	32	1
Все команды условного перехода, кроме JCXZ:		
Jccc близкая-метка	16 или 4	2
JCXZ близкая-метка	18 или 6	2
JMP близкая-метка	15	2
JMP метка-атрибут NEAR	15	3
JMP метка-атрибут FAR	15	5
JMP указатель-память 16	18+EA	2—4
JMP указатель-регистр 16	11	2
JMP указатель-память 32	24+EA	2—4
LAHF	4	1
LDS регистр 16, память 32	24+EA	2—4
LEA регистр 16, память 16	2+EA	2—4
LES регистр 16, память 32	24+EA	2—4
LOCK	2	1
LODS строка_источник	12 (16)	1
LODS (повтор) строка_источник	9+13 (17)/повтор	1
LOOP близкая_метка	17 или 5	2
LOOPE/LOOPZ близкая_метка	18 или 6	2
LOOPNE/LOOPNZ близкая_метка	19 или 5	2
MOV память, аккумулятор	10 (14)	3
MOV аккумулятор, память	10 (14)	3
MOV регистр, регистр	2	2
MOV регистр, память	8 (12)+EA	2—4
MOV память, регистр	9 (13)+EA	2—4
MOV регистр, непосредственный-операнд	4	2—3
MOV память, непосредственный-операнд	10 (14)+EA	3—6
MOV регистр-сегмента, регистр 16	2	2

Команда	Число тактов	Число байтов
MOV регистр-сегмента, память 16	8 (12)+EA	2–4
MOV регистр 16, регистр-сегмента	2	2
MOV память 16, регистр-сегмента	9 (13)+EA	2–4
MOVS строка_приемник, строка_источник	18 (26)	1
MOVS (повтор) строка_приемник, строка_источник	9+17 (25)/повтор	1
MUL регистр 8	70–77	2
MUL регистр 16	118–133	2
MUL память 8	(76–83)+EA	2–4
MUL память 16	(128–143)+EA	2–4
NEG регистр	3	2
NEG память	16 (24)+EA	2–4
NOP	3	1
NOT регистр	3	2
NOT память	16 (24)+EA	2–4
OR регистр, регистр	3	2
OR регистр, память	9 (13)+EA	2–4
OR память, регистр	16 (24)+EA	2–4
OR регистр, непосредственный-операнд	4	3–4
OR память, непосредственный-операнд	17 (15)+EA	3–6
OR аккумулятор, непосредственный-операнд	4	2–3
OUT непосредственный-операнд 8, аккумулятор	10 (14)	2
OUT DX, аккумулятор	8 (12)	1
POP регистр	12	1
POP регистр-сегмента (CS недопустим)	12	1
POP память	25+EA	2–4
POPF	12	1
PUSH регистр	15	1
PUSH регистр-сегмента (CS недопустим)	14	1
PUSH память	24+EA	2–4
PUSHF	14	1
RCL регистр, 1	2	2
RCL регистр, CL	8+4/бит	2
RCL память, 1	15 (23)+EA	2–4
RCL память, CL	20 (28)+EA+4/бит	2–4
RCR регистр, 1	2	2
RCR регистр, CL	8+4/бит	2
RCR память, 1	15 (23)+EA	2–4
RCR память, CL	20 (28)+EA+4/бит	2–4
REP	2	1
REPE/REPZ	2	1

Команда	Число тактов	Число байтов
REPNE/REPNZ	2	1
RET (внутри сегмента, без удаления значений из стека)	20	1
RET (внутри сегмента, с удалением значений из стека)	24	3
RET (между сегментами, без удаления значений из стека)	32	1
RET (между сегментами, с удалением значений из стека)	31	3
ROL регистр, 1	2	2
ROL регистр, CL	8+4/бит	2
ROL память, 1	15 (23)+EA	2–4
ROL память, CL	20 (28)+EA+4/бит	2–4
ROR регистр, 1	2	2
ROR регистр, CL	8+4/бит	2
ROR память, 1	15 (23)+EA	2–4
ROR память, CL	20 (28)+EA+4/бит	2–4
SAHF	4	1
SAL/SHL регистр, 1	2	2
SAL/SHL регистр, CL	8+4/бит	2
SAL/SHL память, 1	15 (23)+EA	2–4
SAL/SHL память, CL	20 (28)+EA+4/бит	2–4
SAR регистр, 1	2	2
SAR регистр, CL	8+4/бит	2
SAR память, 1	15 (23)+EA	2–4
SAR память, CL	20 (28)+EA+4/бит	2–4
SBB регистр, регистр	3	2
SBB регистр, память	9 (13)+EA	2–4
SBB память, регистр	16 (24)+EA	2–4
SBB регистр, непосредственный-операнд	4	3–4
SBB память, непосредственный-операнд	17 (25)+EA	3–6
SBB аккумулятор, непосредственный-операнд	4	2–3
SCAS строка_приемник	15 (19)	1
SCAS (повтор) строка_приемник	9+15 (19)/повтор	1
SHR регистр, 1	2	2
SHR регистр, CL	8+4/бит	2
SHR память, 1	15 (23)+EA	2–4
SHR память, CL	20 (28)+EA+4/бит	2–4
STC	2	1
STD	2	1
STI	2	1
STOS строка_приемник	11 (15)	1
STOS (повтор) строка_приемник	9+10 (14)/повтор	1
SUB регистр, регистр	3	2
SUB регистр, память	9 (13)+EA	2–4
SUB память, регистр	16 (24)+EA	2–4

Команда	Число тактов	Число байтов
SUB регистр, непосредственный-операнд	4	3–4
SUB память, непосредственный-операнд	17 (25)+EA	3–6
SUB аккумулятор, непосредственный-операнд	4	2–3
TEST регистр, регистр	3	2
TEST регистр, память	9 (13)+EA	2–4
TEST регистр, непосредственный-операнд	4	3–4
TEST память, непосредственный-операнд	10 (14)+EA	3–6
TEST аккумулятор, непосредственный-операнд	4	2–3
WAIT	3+5n	1
XCHG аккумулятор, регистр 16	3	1
XCHG память, регистр	17 (25)+EA	2–4
XCHG регистр, регистр	4	2
XLAT таблица_источник	11	1
XOR регистр, регистр	3	2
XOR регистр, память	9 (13)+EA	2–4
XOR память, регистр	16 (24)+EA	2–4
XOR регистр, непосредственный-операнд	4	3–4
XOR память, непосредственный-операнд	17 (15)+EA	3–6
XOR аккумулятор, непосредственный-операнд	4	2–3

2. Для большинства из тех команд, которые обращаются к памяти, в столбце циклов добавлена аббревиатура EA (Effective Address – исполнительный адрес). Она сообщает, что для вычисления исполнительного адреса требуются дополнительные циклы; требуемое на это время указано в табл. В.1.

3. Времена исполнения команд условной передачи управления и команд управления циклами зависят от того, должен ли быть выполнен переход. Если переход выполняется, то берите из столбца циклов большее число. В противном случае, если управление "проскакивает" к следующей команде, берите меньшее число.

Предположим, например, что Вам требуется найти время исполнения команды

```
ADD ES:[BX],DX
```

Общая форма этой команды – ADD **память, регистр**, поэтому ее время исполнения дается соотношением

9 (24) + EA

Так как в данном случае команда ADD оперирует словами, то надо взять не 9, а 24. Поскольку в соотношение входит EA, то надо добавить время вычисления адреса, взяв его из табл. В.1. Операнд-приемник имеет форму [BX], по которой находим, что надо добавить пять дополнительных циклов. Но поскольку в опе-

ранд входит указание замены сегмента (ES:), то надо добавить еще два цикла. После сложения всех трех значений получаем, что полное исполнение равно 31 циклу тактового генератора (24+5+2).

ПРИЛОЖЕНИЕ Г. СИСТЕМА КОМАНД МИКРОПРОЦЕССОРА 8088

В табл. Г.1 приведены команды микропроцессора 8088 в алфавитном порядке. Для каждой команды показан ее общий формат на языке ассемблера и указано, на какие флаги она воздействует. В столбце "Флаги" знак "–" означает, что состояние флага сохраняется, знак "*" – что флаг может изменить состояние, знак "?" – что состояние флага становится неопределенным.

Таблица Г.1. **Система команд микропроцессора 8088**

Мнемокод	Формат	Флаги								
		OF	DF	IF	TF	SF	ZF	AF	PF	CF
AAA	AAA	?	–	–	–	?	?	*	?	*
AAD	AAD	?	–	–	–	*	*	?	*	?
AAM	AAM	?	–	–	–	*	*	?	*	?
AAS	AAS	?	–	–	–	?	?	*	?	*
ADC	ADC приемник, источник	*	–	–	–	*	*	*	*	*
ADD	ADD приемник, источник	*	–	–	–	*	*	*	*	*
AND	AND приемник, источник	0	–	–	–	*	*	?	*	0
CALL	CALL имя	–	–	–	–	–	–	–	–	–
CBW	CBW	–	–	–	–	–	–	–	–	–
CLC	CLC	–	–	–	–	–	–	–	–	0
CLD	CLD	–	0	–	–	–	–	–	–	–
CLI	CLI	–	–	0	–	–	–	–	–	–
CMC	CMC	–	–	–	–	–	–	–	–	*
CMP	CMP приемник, источник	*	–	–	–	*	*	*	*	*
CMPS	CMPS строка_приемник, строка_источник	*	–	–	–	*	*	*	*	*
CMPSB	CMPSB	*	–	–	–	*	*	*	*	*
CMPSW	CMPSW	*	–	–	–	*	*	*	*	*
CWD	CWD	–	–	–	–	–	–	–	–	–
DAA	DAA	?	–	–	–	*	*	*	*	*
DAS	DAS	?	–	–	–	*	*	*	*	*
DEC	DEC приемник	*	–	–	–	*	*	*	*	–
DIV	DIV источник	?	–	–	–	?	?	?	?	?
ESC	ESC код_внеш_оп, источник	–	–	–	–	–	–	–	–	–
HLT	HLT	–	–	–	–	–	–	–	–	–
IDIV	IDIV источник	?	–	–	–	?	?	?	?	?

Мнемокод	Формат	Флаги								
		OF	DF	IF	TF	SF	ZF	AF	PF	CF
IMUL	IMUL источник	*	–	–	–	?	?	?	?	*
IN	IN аккумулятор, порт	–	–	–	–	–	–	–	–	–
INC	INC приемник	*	–	–	–	*	*	*	*	–
INT	INT тип_прерывания	–	–	0	0	–	–	–	–	–
INTO	INTO	–	–	0	0	–	–	–	–	–
IRET	IRET	*	*	*	*	*	*	*	*	*
JA/JNBE	JA близкая_метка	–	–	–	–	–	–	–	–	–
JAE/JNB	JAE близкая_метка	–	–	–	–	–	–	–	–	–
JB/JNAE/JC	JB близкая_метка	–	–	–	–	–	–	–	–	–
JBE/JNA	JBE близкая_метка	–	–	–	–	–	–	–	–	–
JCXZ	JCXZ близкая_метка	–	–	–	–	–	–	–	–	–
JE/JZ	JE близкая_метка	–	–	–	–	–	–	–	–	–
JG/JNLE	JG близкая_метка	–	–	–	–	–	–	–	–	–
JGE/JNL	JGE близкая_метка	–	–	–	–	–	–	–	–	–
JL/JNGE	JL близкая_метка	–	–	–	–	–	–	–	–	–
JLE/JNG	JLE близкая_метка	–	–	–	–	–	–	–	–	–
JMP	JMP имя	–	–	–	–	–	–	–	–	–
JNC	JNC близкая_метка	–	–	–	–	–	–	–	–	–
JNE/JNZ	JNE близкая_метка	–	–	–	–	–	–	–	–	–
JNO	JNO близкая_метка	–	–	–	–	–	–	–	–	–
JNP/JPO	JNP близкая_метка	–	–	–	–	–	–	–	–	–
JNS	JNS близкая_метка	–	–	–	–	–	–	–	–	–
JO	JO близкая_метка	–	–	–	–	–	–	–	–	–
JP/JPE	JP близкая_метка	–	–	–	–	–	–	–	–	–
JS	JS близкая_метка	–	–	–	–	–	–	–	–	–
LAHF	LAHF	–	–	–	–	–	–	–	–	–
LDS	LDS регистр 16, память 32	–	–	–	–	–	–	–	–	–
LEA	LEA регистр 16, память 16	–	–	–	–	–	–	–	–	–
LES	LES регистр 16, память 32	–	–	–	–	–	–	–	–	–
LOCK	LOCK	–	–	–	–	–	–	–	–	–
LODS	LODS строка_источник	–	–	–	–	–	–	–	–	–
LODSB	LODSB	–	–	–	–	–	–	–	–	–
LODSW	LODSW	–	–	–	–	–	–	–	–	–
LOOP	LOOP близкая_метка	–	–	–	–	–	–	–	–	–
LOOPE/LOOPZ	LOOPE близкая_метка	–	–	–	–	–	–	–	–	–
LOOPNE/LOOPNZ	LOOPNE близкая_метка	–	–	–	–	–	–	–	–	–
MOV	MOV приемник, источник	–	–	–	–	–	–	–	–	–
MOVS	MOVS строка_приемник, строка_источник	–	–	–	–	–	–	–	–	–
MOVSB	MOVSB	–	–	–	–	–	–	–	–	–
MOVSW	MOVSW	–	–	–	–	–	–	–	–	–

Мнемокод	Формат	Флаги								
		OF	DF	IF	TF	SF	ZF	AF	PF	CF
MUL	MUL источник	*	–	–	–	?	?	?	?	*
NEG	NEG приемник	*	–	–	–	*	*	*	*	*
NOP	NOP	–	–	–	–	–	–	–	–	–
NOT	NOT приемник	–	–	–	–	–	–	–	–	–
OR	OR приемник, источник	0	–	–	–	*	*	?	*	0
OUT	OUT порт, аккумулятор	–	–	–	–	–	–	–	–	–
POP	POP приемник	–	–	–	–	–	–	–	–	–
POPF	POPF	*	*	*	*	*	*	*	*	*
PUSH	PUSH источник	–	–	–	–	–	–	–	–	–
PUSHF	PUSHF	–	–	–	–	–	–	–	–	–
RCL	RCL приемник, счетчик	*	–	–	–	–	–	–	–	*
RCR	RCR приемник, счетчик	*	–	–	–	–	–	–	–	*
REP	REP	–	–	–	–	–	–	–	–	–
REPE/REPZ	REPE	–	–	–	–	–	–	–	–	–
REPNE/REPNZ	REPNE	–	–	–	–	–	–	–	–	–
RET	RET [число удаляемых из стека значеиий]	–	–	–	–	–	–	–	–	–
ROL	ROL приемник, счетчик	*	–	–	–	–	–	–	–	*
ROR	ROR приемник, счетчик	*	–	–	–	–	–	–	–	*
SAHF	SAHF	–	–	–	–	*	*	*	*	*
SAL/SHL	SAL приемник, счетчик	*	–	–	–	*	*	?	*	*
SAR	SAR приемник, счетчик	*	–	–	–	*	*	?	*	*
SBB	SBB приемник, источник	*	–	–	–	*	*	*	*	*
SCAS	SCAS строка_приемник	*	–	–	–	*	*	*	*	*
SCASB	SCASB	*	–	–	–	*	*	*	*	*
SCASW	SCASW	*	–	–	–	*	*	*	*	*
SHR	SHR приемник, счетчик	*	–	–	–	0	*	?	*	*
STC	STC	–	–	–	–	–	–	–	–	1
STD	STD	–	1	–	–	–	–	–	–	–
STI	STI	–	–	1	–	–	–	–	–	–
STOS	STOS строка_приемник	–	–	–	–	–	–	–	–	–
STOSB	STOSB	–	–	–	–	–	–	–	–	–
STOSW	STOSW	–	–	–	–	–	–	–	–	–
SUB	SUB приемник, источник	*	–	–	–	*	*	*	*	*
TEST	TEST приемник, источник	0	–	–	–	*	*	?	*	0
WAIT	WAIT	–	–	–	–	–	–	–	–	–
XCHG	XCHG приемник, источник	–	–	–	–	–	–	–	–	–
XLAT	XLAT таблица_источник	–	–	–	–	–	–	–	–	–
XOR	XOR приемник, источник	0	–	–	–	*	*	?	*	0

ПРИЛОЖЕНИЕ Д. РУКОВОДСТВО ПО ПОЛЬЗОВАНИЮ ДИСКОМ[1]

Диск, который можно заказать с этой книгой, содержит около 40 приведенных в ней программ. Это двусторонний диск, "перевертыш": чтобы воспользоваться программами, заполненными на его обратной стороне, диск надо вставить обратной стороной вверх. Верхняя сторона содержит *исходные программы,* включая модели, представленные в гл. 2 (MAINMOD и SECMOD), а также макробиблиотеку MACRO.LIB, содержащую свыше 30 "мини-программ" или *макроопределений.* Обратная сторона диска содержит оттранслированные *объектные программы.*

Наличие объектных программ позволяет Вам загружать их и исполнять, не выполняя трансляцию.

НЕПОСРЕДСТВЕННО ИСПОЛНЯЕМЫЕ ПРОГРАММЫ

Некоторые из приведенных в этой книге программ рассчитаны на самостоятельное исполнение без привлечения других программ. Для каждой из этих программ, перечисленных в табл. Д.1, диск содержит *исполняемый файл* в дополнение к исходному и объектному файлам. Исполняемые файлы, имена которых имеют расширение EXE, расположены на той же стороне диска, что и объектные программы.

Таблица Д.1. **Программы, для которых есть исполняемые файлы (типа EXE)**

Имя файла	Пример в книге[1]	Описание
DIAGLINE	7.1	Изображение диагонали
HEX2DEC	–	Преобразование шестнадцатеричного числа в десятичное
KEYMUSIC	8.5	Исполнение музыки с клавиатуры
LOCK	6.5	Защита файла от записи
MARY	8.3	Исполнение мелодии "У Мэри была маленькая овечка"
MOVEBIRD	(7.1)	Движение "птички" по экрану
MOVEFACE	7.2	Движение "улыбающейся рожицы" по экрану
NEWDIAG	9.1	Изображение диагонали с использованием макроопределений
RANDFACE	(7.2)	Случайное движение "улыбающейся рожицы" по экрану
TURKEY	8.4	Исполнение мелодии "Индюк в соломе"
UNLOCK	6.6	Снятие с файла защиты от записи

[1] Числа в скобках относятся к ответам на упражнения. Например (7.1) означает ответ к упр. 1 гл.7

[1] В этом приложении речь идет о диске, поставляемом с оригиналом книги, а не с ее переводом. Однако внимательиому читателю не составит особого труда самому подготовить такой диск в учебных целях. – *Прим. перев.*

Как только операционная система DOS выдаст на экран приглашение к вводу (A> или C>), Вы можете вызвать любую из этих программ, просто набрав ее имя. Например, чтобы проиграть мелодию "Индюк в соломе", надо набрать turkey (или b:turkey, если диск с данными вставлен в дисковод B).

Диск содержит также исходный файл для примера программы EX_PROG, приведенного в гл. 2, но для нее на нем нет ни объектного, ни исполняемого файла. Вы сами должны создать эти файлы, используя описанную ниже процедуру.

ЗАГРУЖАЕМЫЕ ФАЙЛЫ

Остальные файлы на диске представляют собой процедуры общего назначения, которые Вы можете использовать в своих программах. Однако *сначала Вам надо создать вызывающую программу,* оттранслировать ее и загрузить два объектных файла (с расширением OBJ), чтобы получить исполняемый файл (с расширением EXE). В табл. Д.2 содержится информация, которая понадобится Вам для работы с этими файлами.

Правда, некоторые процедуры из этой книги вызывают другие процедуры. Для Вашего удобства помещенные на диск программы *включают* все процедуры, которые ими вызываются. Например, процедура MUL32 из примера 4.2 вызывает процедуру MULU32 из примера 4.1. Поэтому объектный файл MULS32.OBJ, содержащий процедуру MULS32, включает и команды процедуры MULU32. Таким образом, Вам никогда не понадобится загружать более двух объектных файлов: достаточно иметь вызывающую программу и программу, выбранную Вами на диске.

Таблица Д.2. **Программы, которые требуют загрузки**

Загружаемый объектный модуль	Вызываемая процедура	Пример[1] в книге	Описание
ADD_2_OL	ADD_TO_OL	5.7	Добавление элемента к упорядоченному списку
ADD2UL	ADD_TO_UL	(5.1)	Создание неупорядоченного списка
ADD_2_UL	ADD_TO_UL	5.1	Добавление элемента к неупорядоченному списку
ASC_BIN	ASCII_BIN	6.7	Преобразование строки в двоичное число
AVERAGE	AVERAGE	4.3	Усреднение слов без знака
B_SEARCH	B_SEARCH	5.6	Поиск в упорядоченном списке
BIN_ASC	BIN_ASCII	6.8	Преобразование двоичного числа в строку
BUBBLE	BUBBLE	5.5	Пузырьковая сортировка неупорядоченного списка
CLR_S	CLEAR_SCREEN	7.3	Стирание строк на экране
CONV_HEX	CONV_HEX	5.11	Преобразование шестнадцатиричной цифры в коды ASCII, BCD и EBCDIC
COSINE	FIND_COS	5.10	Вычисление косинуса угла
D_SHAPE	DSPLY_SHAPE	7.4	Формирование изображения с помощью таблицы образа

Загружаемый объектный модуль	Вызываемая процедура	Пример в книге[1]	Описание
DEL_OL	DEL_OL	5.8	Удаление элемента из упорядоченного списка
DEL_UL	DEL_UL	5.2	Удаление элемента из неупорядоченного списка
DELAY	DELAY	6.3	Пауза заданной длительности
DIVUO	DIVUO	4.4	Деление с учетом переполнения
M_SHARE	MOVE_SHARE	7.5	Движение образа по экрану
MINMAX	MINMAX	5.3	Поиск максимума и минимума в неупорядоченном списке
MULS32	MULS32	4.2	Умножение 32-битовых чисел со знаком
MULU32	MULU32	4.1	Умножение 32-битовых чисел без знака
PHONE_#	PHONE_NOS	5.13	Сортировка телефонного справочника
PLAY	PLAY	8.2	Исполнение мелодии
RAND_51	RAND_51	6.1	Генерация случайного числа в диапазоне от 0 до 51
READKEYS	READ_KEYS	6.2	Чтение строки с клавиатуры
REPLACE	REPLACE	(5.2)	Замеиа элемента в упорядоченном списке
SHOW_ERR	SHOW_ERR	6.4	Изображение сообщений об ошибках при вызове функций DOS 2
SINE	FIND_SINE	5.9	Вычисление синуса угла
SOUND	SOUND	8.1	Генерация звука
SQRT32	SQRT32	4.5	Вычисление квадратного корня методом Ньютона
SR32	SR32	(4.1)	Вычисление квадратного корня путем последовательных вычитаний
TIMEKEYS	TIME_KEYS	(6.1)	Определение времени между нажатиями на клавиши

[1] Числа в скобках относятся к ответам на упражнения. Например (5.1) означает ответ к упр. 1 гл.5.

СОЗДАНИЕ ВЫЗЫВАЮЩЕЙ ПРОГРАММЫ

Вызывающая программа представляет собой процедуру, которая исполняет одну или несколько процедур, а затем возвращает управление системной программе, использованной Вами для ее исполнения (операционной системе DOS или отладчику DEBUG). Каждая вызывающая программа должна содержать сегмент команд, сегмент стека и псевдооператор EXTRN.

Сегмент команд содержит команду CALL (например, CALL MULU32), а также команду RET, обеспечивающую возврат управления операционной системе DOS. Далее, нередко в нем находятся команды присваивания начальных значений тем регистрам и ячейкам памяти, через которые вызывающая программа рассчитывает получить результат.

Сегмент стека резервирует в памяти место для адресов возврата.

Псевдооператор EXTRN сообщает Ассемблеру, что процедура, вызываемая командой CALL из сегмента команд, находится на диске в другом объектном файле. Например, оператор EXTRN MULU32:FAR сообщает Ассемблеру, что процедура MULU32 находится в объектном файле, который надо загрузить вместе с текущим файлом; суффикс FAR сообщает ему, что процедура MULU32 находится в другом сегменте команд.

Если для вызова процедуры требуется данные в сегменте данных или в дополнительном сегменте, то Ваша вызывающая программа должна иметь и эти сегменты.

После ввода текста вызывающей программы в ЭВМ и сохранения ее на диске надо оттранслировать ее и загрузить два объектных модуля для создания исполняемого модуля. На рис. Д.1 показан листинг программы, вызывающей процедуру умножения 32-битовых чисел без знака MULU32, взятую из примера 4.1. Оттранслируйте эту процедуру (файл MULU32.ASM), чтобы получить объектный файл MULU32.OBJ, а затем исполните команду

```
B>a:link mulu32c+mulu32;
```

для создания исполняемого модуля MULU32.EXE, который можно исполнить под управлением отладчика DEBUG.

Показанная на рис. Д.1 программа имеет общее назначение; она не сообщает процедуре MULU32, какие числа ей надо перемножить. После вызова отладчика DEBUG Вам надо воспользоваться командой R (register – регистр) для загрузки этих чисел в регистры CX, BX (множимое) и DX, AX (множитель).

Так как программа MULU32C не дает видимых результатов, то лучший способ ее исполнения состоит в том, чтобы дать ей возможность дойти до команды RET, а затем проверить содержимое регистров. Если Вы оттранслируете программу MULU32C и изобразите (командой type) листинговый файл (MULU32C.LST), то обнаружите, что команда RET имеет смещение адреса 000A. Следовательно, Вам надо дать отладчику DEBUG команду ga (go to A – перейти к A).

```
TITLE   MULU32C - вызывающая программа для MULU32
        PAGE        ,132
        EXTRN       MULU32:FAR    ;MULU32 - внешняя процедура
;
;  Определить сегмент стека
;
STACK   SEGMENT     PARA STACK  STACK'
        DB          64 DUP('STACK     )
STACK   ENDS
;
;  Команды вызывающей программы
;
CSEG    SEGMENT     PARA PUBLIC  CODE
CALLER  PROC        FAR
        ASSUME      CS:CSEG,SS:STACK
;
;  Поместить в стек такие начальные значения, чтобы программа
:  могла возвратить управление отладчику DEBUG
;
        PUSH        DS      ;Поместить номер блока адреса возврата
;                             в стек
```

```
          MOV       DI,0   ;Обнулить регистр
          PUSH      DI     ;Поместить в стек нулевой адрес возврата
;
;   Вызвать MULU32
;
          CALL      MULU32
          RET              ;Возвратиться к отладчику DEBUG
CALLER    ENDP
CSEG      ENDS
          END       CALLER
```

Рис. Д.1. Вызывающая программа для процедуры MULU32

УКАЗАТЕЛЬ ТЕРМИНОВ

УКАЗАТЕЛЬ КОМАНД И ПСЕВДООПЕРАТОРОВ

КОМАНДЫ

ПСЕВДООПЕРАТОРЫ

IBM PC & XT Assembly Language

A Guide for Programmers

Enhanced and Enlarged

Leo J. Scanlon

Brady Communications Company, Inc.
A Simon&Schuster Publishing Company
New York, NY 10020

Л. Скэнлон

Персональные ЭВМ IBM PC и XT

Программирование на языке ассемблера

Перевел с английского И. В. ЕМЕЛИН



Москва
«Радио и связь»
1989